

Aztec C Library Manual for the Amiga

**Version 5.0
October 1989**

Copyright © 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

PUBLISHED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

USE RESTRICTIONS

You are permitted to install and use this product on a single computer. Multiple CPU systems require supplementary licenses.

Before using any Aztec C products, the License Registration included with this product must be signed and mailed to:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702

COPYRIGHT

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to this product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to modify the programs and revise the contents of the manual without obligation to notify any person of such revision or changes.

TRADEMARKS

Aztec C, Manx AS, Manx LN, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh and Apple II are trademarks of Apple Computer. Atari is a trademark of Atari Computers. Amiga is a trademark of Commodore-Amiga.

Manual Revision History

October 1989	Fifth Edition
December 1988	Fourth Edition
May 1988	Third Edition
April 1986	Second Edition
February 1986	First Edition

Table of Contents

Chapter 1 - Introduction

ANSI C	1 - 1
--------------	-------

Chapter 2 - Library Overview

Introduction	2 - 1
Library Summary	2 - 1
31-Character Names	2 - 2
Integer Sizes	2 - 2
Overview of Aztec C and Amiga I/O	2 - 3
Aztec C I/O Functions	2 - 3
Preopened Devices and Command Line Arguments	2 - 4
File I/O	2 - 5
Sequential I/O	2 - 5
Random I/O	2 - 5
Opening Files	2 - 6
Device I/O	2 - 6
Console I/O	2 - 6
I/O to Other Devices	2 - 7
Mixing Unbuffered and Standard I/O Calls	2 - 7
AMIGA I/O FUNCTIONS	2 - 7
Preopened Devices and Command Line Arguments	2 - 8
FILE I/O	2 - 8
Device I/O	2 - 8
Aborting a Program from the Console	2 - 8
Overview of Aztec Standard I/O	2 - 9
Aztec C Standard I/O Functions	2 - 9
Opening Files and Devices	2 - 9

Closing Streams	2 - 10
Sequential I/O	2 - 10
Random I/O	2 - 10
Buffering	2 - 11
Errors	2 - 11
THE STANDARD I/O FUNCTIONS	2 - 12
AMIGA STANDARD I/O	2 - 14
Buffering	2 - 14
Overview of Aztec C and Amiga Unbuffered I/O	2 - 14
FILE I/O	2 - 16
DEVICE I/O	2 - 17
Unbuffered I/O to the Console	2 - 17
Unbuffered I/O to Nonconsole Devices	2 - 17
Overview of Aztec C and Amiga Console I/O	2 - 17
Character-Oriented Input	2 - 17
Writing System-Independent Programs	2 - 18
Using ioctl	2 - 18
THE sgty FIELD	2 - 19
EXAMPLE	2 - 19
Console Input - RAW Mode	2 - 19
Overview of Aztec C and Amiga Dynamic	
Buffer Allocation	2 - 20
Dynamic Allocation of Standard I/O Buffers	2 - 21
Overview of Dynamic Buffer Allocation	2 - 21
Overview of Aztec C and Amiga Error Processing	2 - 21
Overview of ANSI Header Files	2 - 23
assert.h	2 - 23
ctype.h	2 - 24
errno.h	2 - 24
fcntl.h	2 - 24

float.h	2 - 24
limits.h	2 - 24
locale.h	2 - 24
math.h	2 - 24
setjmp.h	2 - 24
signal.h	2 - 24
stdarg.h	2 - 24
stddef.h	2 - 25
stdio.h	2 - 25
stdlib.h	2 - 25
string.h	2 - 25
time.h	2 - 25

Chapter 3 - Library Functions

Introduction	3 - 1
FUNCTION LIST	3 - 2
DESCRIPTION FORMAT	3 - 6
_abort (C) - Miscellaneous	3 - 9
abs (C) - Integer Math	3 - 10
access (C) - Standard I/O	3 - 11
acos (M) - Float Math	3 - 13
asctime (C) - Time	3 - 14
asin (M) - Float Math	3 - 15
assert (Macro) - Miscellaneous	3 - 16
atan (M) - Float Math	3 - 17
atan2 (M) - Float Math	3 - 18
atexit (C) - Miscellaneous	3 - 19

atof (M) - Conversion	3 - 20
atoi(C) - Conversion	3 - 21
atol (C) - Conversion	3 - 22
calloc (C) - Memory Allocation	3 - 23
ceil (M) - Float Math	3 - 24
clearerr (C) - Standard I/O	3 - 25
_cli_parse (C) - Amiga	3 - 26
clock (C) - Time	3 - 27
close (C) - UNIX I/O	3 - 28
cos (M) - Float Math	3 - 29
cosh (M) - Float Math	3 - 30
cotan (M) - Float Math	3 - 31
creat (C) - UNIX I/O	3 - 32
ctime (C) - Time	3 - 33
difftime (C) - Time	3 - 34
div (C) - Integer Math	3 - 35
dos_packet (C) - Amiga	3 - 36
exec (C) - Amiga	3 - 37
Parameters	3 - 37
The Functions	3 - 38
Other Features	3 - 38
exit (C) - Miscellaneous	3 - 39
exp (M) - Float Math	3 - 40
fabs (M) - Float Math	3 - 41
fclose (C) - Standard I/O	3 - 42

fdopen (C) - UNIX I/O	3 - 43
feof (C) - Standard I/O	3 - 46
ferror (C) - Standard I/O	3 - 47
fexecl, fexecv(C) - Amiga	3 - 48
Parameters	3 - 48
The Functions	3 - 49
Other Information	3 - 49
fflush (C) - Standard I/O	3 - 51
fgetc (C) - Standard I/O	3 - 52
fgetpos (C) - Standard I/O	3 - 53
fgets (C) - Standard I/O	3 - 54
fileno (C) - UNIX I/O	3 - 55
floor (M) - Float Math	3 - 56
fmod (M) - Float Math	3 - 57
fopen (C) - Standard I/O	3 - 58
format (C) - Conversion	3 - 61
fprintf (C, M) - Standard I/O	3 - 62
fputc (C) - Standard I/O	3 - 63
fputs (C) - Standard I/O	3 - 64
fread (C) - Standard I/O	3 - 65
free (C) - Memory Allocation	3 - 66
freeseq (C) - Amiga	3 - 67
freopen (C) - Standard I/O	3 - 68
frexp (M) - Float Math	3 - 69
fscanf (C,M) - Standard I/O	3 - 70

fseek (C) - Standard I/O	3 - 71
fsetpos (C) - Standard I/O	3 - 73
ftell (C) - Standard I/O	3 - 74
ftoa (M) - Conversion	3 - 75
fwrite (C) - Standard I/O	3 - 76
geta4 (C) - Amiga	3 - 78
getc (C) - Standard I/O	3 - 79
getchar (C) - Standard I/O	3 - 80
getenv (C) - Miscellaneous	3 - 81
gets (C) - Standard I/O	3 - 82
gmtime (C) - Time	3 - 83
index (C) - String Handling	3 - 84
int_end (C) - Amiga	3 - 85
int_start (C) - Amiga	3 - 86
ioctl (C) - UNIX I/O	3 - 87
is... (C) - Character Type	3 - 88
labs (C) - Integer Math	3 - 90
ldexp(M) - Float Math	3 - 91
ldiv (C) - Integer Math	3 - 92
localtime (C) - Time	3 - 93
log (M) - Float Math	3 - 94
log10 (M) - Float Math	3 - 95
longjmp (C) - Miscellaneous	3 - 96
lseek (C) - UNIX I/O	3 - 97
malloc (C) - Memory Allocation	3 - 99

memchr (C) - Block Operation	3 - 100
memcmp (C) - Block Operation	3 - 101
memcpy (C) - Block Operation	3 - 102
memmove (C) Block Operation	3 - 103
memset (C) - Block Operation	3 - 104
mktemp (C) - UNIX I/O	3 - 105
mktime (C) - Time	3 - 107
modf (M) - Float Math	3 - 108
movmem(C) Block Operation	3 - 109
open (C) - UNIX I/O	3 - 110
perror (C) - Standard I/O	3 - 113
pow (M) - Float Math	3 - 115
printf (C, M) - Standard I/O	3 - 116
The Format String	3 - 116
Conversion Specifiers	3 - 116
Flags Field	3 - 117
Width Field	3 - 118
Precision Field	3 - 118
Size-mod Field	3 - 118
Type Field	3 - 119
putc (C) - Standard I/O	3 - 122
putchar (C) - Standard I/O	3 - 123
puts (C) - Standard I/O	3 - 124
qsort (C) - Miscellaneous	3 - 125
ran (M) - Float Math	3 - 127
rand (C) - Integer Math	3 - 128

read (C) - UNIX I/O	3 - 129
realloc (C) - Memory Allocation	3 - 130
remove (C) - Standard I/O	3 - 131
rename (C) - Standard I/O	3 - 132
rewind (C) - Standard I/O	3 - 133
rindex (C) - String Handling	3 - 134
sbrk (C) - Memory Allocation	3 - 135
scanf (C,M) - Standard I/O	3 - 136
THE FORMAT STRING	3 - 136
Matching White Space Characters	3 - 136
Matching Ordinary Characters	3 - 137
Matching Conversion Specifications	3 - 137
Details of Input Conversion	3 - 137
scdire (C) - Amiga	3 - 141
screen (S) - Amiga	3 - 142
segload (C) - Amiga	3 - 144
setbuf (C) - Standard I/O	3 - 145
setenv (C) - Miscellaneous	3 - 146
setjmp - Miscellaneous	3 - 147
setvbuf (C) - Standard I/O	3 - 148
signal (C) - Signal	3 - 150
Signals and the sig Parameter	3 - 150
Signal Processing and the func Parameter	3 - 150
Return Values from Signal	3 - 151
sin (M) - Float Math	3 - 153
sinh (M) - Float Math	3 - 154

sprintf (C, M) - Conversion	3 - 155
sqrt (M) - Float Math	3 - 156
srn (M) - Float Math	3 - 157
srand (C) - Integer Math	3 - 158
sscanf (C,M) - Conversion	3 - 159
stat (C) - UNIX I/O	3 - 160
_stkchk (C) - Miscellaneous	3 - 161
_stkover (C) - Miscellaneous	3 - 162
strcat (C) - String Handling	3 - 163
strchr (C) - String Handling	3 - 164
strcmp (C) - String Handling	3 - 165
strcoll (C) - String Handling	3 - 167
strcpy (C) - String Handling	3 - 168
strcspn (C) - String Handling	3 - 169
strlen (C) - String Handling	3 - 170
strncat (C) - String Handling	3 - 171
strncmp (C) - String Handling	3 - 172
strncpy (C) - String Handling	3 - 173
strpbrk (C) - String Handling	3 - 174
strrchr (C) - String Handling	3 - 175
strspn (C) - String Handling	3 - 176
strstr (C) - String Handling	3 - 177
strtod (C) - Conversion	3 - 178
strtok (C) - String Handling	3 - 179
strxfrm (C) - String Handling	3 - 181

tan (M) - Float Math	3 - 182
tanh (M) - Float Math	3 - 183
time (C) - Time	3 - 184
tmpfile (C) - Standard I/O	3 - 185
tmpnam (C) - Standard I/O	3 - 186
tolower (C) - Conversion	3 - 187
toupper (C) - Conversion	3 - 188
ungetc (C) - Standard I/O	3 - 189
unlink (C) - UNIX I/O	3 - 190
va_arg, va_end, va_start (C) - Variable Arguments	3 - 191
vfprintf (C) - Standard I/O	3 - 192
vprintf (C) - Standard I/O	3 - 193
vsprintf (C) - Conversion	3 - 194
_wb_parse (C) - Amiga	3 - 195
write (C) - UNIX I/O	3 - 196

Chapter 4 - Workbench

Amiga Reference Guides	4 - 1
Amiga Workbench Functions	4 - 2

INTRODUCTION

LIBRARY OVERVIEW

LIBRARY FUNCTIONS

WORKBENCH

INTRODUCTION

1

Chapter 1 - Introduction

This is the *Aztec C Library Manual* and is included as a part of your total Aztec C documentation package. See your *Aztec C User Guide* for a listing of all of the Aztec C configurations and documentation that are available for the Amiga.

The *Aztec C User Guide* showed you how to install your software and how to begin using Aztec C easily and quickly. The *Aztec C Reference Manual* presented more advanced programs and a more thorough understanding of how the Aztec C package works.

This *Aztec C Library Manual* includes an overview, listing, and description of all the library functions that are included with your Aztec C package, and a listing of the Amiga Workbench function names and syntax.

Of course, this *Aztec C Library Manual* also contains a detailed Table of Contents and an Index.

ANSI C

Aztec C is fully dpANSI standard. If you have used Aztec C prior to Version 5.0, you should note that this standardization has caused the library functions included with your Aztec C system to change substantially. Before using Aztec C, you should carefully review the **Library Overview** chapter and the **Library Functions** chapter for a more complete understanding of ANSI and how it has affected the library functions.

Chapter 2 - Library Overview

Introduction

This chapter discusses the libraries included with Aztec C for the Amiga and gives an overview, by category, of the Aztec and Amiga functions provided in this manual.

The chapter first presents a library summary and then divides into the following sections.

1. Overview of Aztec C and Amiga I/O
2. Overview of Aztec C Standard I/O
3. Overview of Aztec C and Amiga Unbuffered I/O
4. Overview of Aztec C and Amiga Console I/O
5. Overview of Aztec C and Amiga Dynamic Buffer Allocation
6. Overview of Aztec C and Amiga Error Processing
7. Overview of ANSI Header Files

Library Summary

More detailed information about Aztec C libraries is available in the **Getting Started** Chapter of the *Aztec C User Guide* . If you are planning to

use special libraries for 16 bit **ints** or specialized floating point libraries, you should read the material in that chapter.

If you are using Amiga Workbench routines, you should read material relating to the Amiga Workbench functions, the **functions.h** header, and direct resident library calls, in the **Getting Started** Chapter of the *Aztec C User Guide* and the **Compiler** Chapter of the *Aztec C Reference Manual*.

Aztec C for the Amiga features support for Workbench functions and includes the following libraries:

c.lib	standard I/O and Amiga toolbox routines
m.lib	Manx IEEE double and extended precision math library
mf.lib	Motorola Fast Floating Point math library
ma.lib	Amiga IEEE double precision math library
m8.lib	68881 coprocessor math library
s.lib	screen functions

The distributed headers and libraries reflect the following:

31-Character Names

Global names in Aztec C have 31 significant characters and have a leading underscore.

Integer Sizes

c.lib uses and expects 32 bit integers. If source files are compiled using 16 bit integers (such as with the **-ps** option) then the 16 bit libraries such as **c16.lib** must be used when linking. Additional information on special libraries and alternate data definitions can be found in the **Getting Started** Chapter of the *Aztec C User Guide*.

Note to previous users: Prior to Version 5.0, Aztec C's default integer size was 16 bits.

Overview of Aztec C and Amiga I/O

There are two sets of functions for accessing files and devices: the unbuffered I/O functions and the standard I/O functions. The Aztec C standard I/O functions conform to the ANSI standard, while the unbuffered I/O functions behave like those described in Chapter 8 of *The C Programming Language*.

Aztec C I/O Functions

The unbuffered I/O functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard I/O functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered I/O functions are used by programs that perform their own blocking and unblocking of disk files. The standard I/O functions are used by programs which need to access files but do not want to be bothered with the details of blocking and unblocking the file records.

The basic procedure for accessing files and devices is the same for both standard and unbuffered I/O: The device or file must first be "opened," e.g., prepared for processing; then I/O operations occur; then the device or file is "closed."

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is 20.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function that performed the open and must be closed by the appropriate function in the same set. There are exceptions to this nonintermingling that are described below.

There are two ways a file or device can be opened: First, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices (standard input, standard output, or standard error) and then opened when the program starts.

Preopened Devices and Command Line Arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character <. Standard output is redirected by entering the name of the file or device, preceded by >.

For example, suppose the executable program **cpy** reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
cpy
```

The following will read from the keyboard and write it to the file **testfile**:

```
cpy > testfile
```

This will copy the file **exmplfil** to the console:

```
cpy < exmplfil
```

And this will copy **exmplfil** to **testfile**:

```
cpy < exmplfil > testfile
```

Aztec C will pass command line arguments to your program via the function **main(argc, argv)**. **argc** is an integer containing the number of arguments plus one; **argv** is a pointer to an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

the program **prog** will be activated and execution begins at your function **main()**. The first parameter to **main** is the integer 4. The second parameter is a pointer to an array of four character pointers; on some systems the first array element will point to the string "**prog**" and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings **arg1**, **arg2**, and **arg3**, respectively.

The command line can contain both arguments to be passed to your program and I/O redirection specifications. The I/O redirection strings will not be passed to your program and can appear anywhere on the command line after the command name but BEFORE any arguments. For example, the standard output of the **prog** program can be redirected to the file **outfile** by the following command; in this case the **argc** and **argv** parameters to the main function of **prog** are the same as if the redirection specifier was not present:

```
prog > outfile arg1 arg2 arg3
```

File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

Sequential I/O

For sequential access, a program simply issues any of the various read or write calls. The transfer will begin at the file's current position and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems that do not keep track of the last character written to a file, it is not always possible to correctly position a file to which data is to be appended.

Random I/O

Two functions are provided which allow a program to set the current position of an open file: **fseek()**, for a file opened for standard I/O; and **lseek()**, for a file opened for unbuffered I/O.

A program accesses a file randomly by first modifying the file's current position using one of the seek functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which do not keep track of the last character written

to a file, positioning relative to the end of a file cannot always be correctly done.

Opening Files

Opening files is somewhat system dependent: The parameters to the open functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of filenames and the areas searched for files differ from system to system.

Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices.

Console I/O

Console I/O can be performed in a variety of ways. There is a default mode, and other modes can be selected by calling the function `ioctl()`.

When the console is in default mode, console input is buffered and is read from the keyboard one line at a time. Typed characters are echoed to the screen and you can use the standard operating system line editing facilities. A program does not have to read an entire line at a time (although the system software does this when reading keyboard input into its internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console I/O allow a program:

- to get characters from the keyboard as they are typed, with or without their being echoed to the display
- to disable normal system line editing facilities
- to terminate a read request if a key is not depressed within a certain interval.

Output to the console is always line buffered: Display occurs when a newline is output or when a request for console input is made. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence.

Optionally, this translation can be disabled.

I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. The options available on the Amiga are discussed in the *Amiga Rom Kernel Reference Manual*.

Mixing Unbuffered and Standard I/O Calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: If a file or device is opened for standard I/O, the function `fileno()` returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device is open for unbuffered I/O, the function `fdopen()` will prepare it for standard I/O as well.

Care is warranted when accessing devices and files with both standard and unbuffered I/O functions.

AMIGA I/O FUNCTIONS

A maximum of 20 files and devices, including the standard I/O devices, can be opened at once for both standard and unbuffered I/O. When this limit is reached, an open file or device must be closed before another can be opened.

Amiga functions are also supplied for accessing files and devices--a program can access some files and devices using the Amiga functions and others using the ANSI-compatible functions.

The ANSI-compatible I/O functions contain tables having a fixed number of entries, one for each open file or device. The number of entries in the table sets the limit on the number of files and devices that can be accessed simultaneously using the ANSI-compatible functions. AmigaDos has similar limits on the number of files and devices that can be accessed. Using an AmigaDos entry will simultaneously tie up an ANSI entry. There are more available AmigaDos entries than ANSI entries. If the ANSI table is full, you might choose to open a device or file using AmigaDos I/O functions.

Preopened Devices and Command Line Arguments

For programs running on an Amiga, the program name is pointed at by the first item in the array that is pointed at by the second argument of the program `main()` function. That is, if the `main()` function begins

```
main(argc, argv)
int argc; char *argv[];
```

the `argv[0]` is a pointer to the program name.

FILE I/O

Unlike some systems supported by Aztec C, positioning of a file relative to its end is always correctly done with the following limitation: You cannot position a file beyond its end.

Device I/O

Using the ANSI-compatible I/O functions, a program can access devices using their AmigaDOS names. For a list of these names, and a description of device I/O using AmigaDOS, see the *AmigaDOS* manuals.

A program can also access devices using Amiga function calls. These calls bypass the ANSI-compatible I/O system and AmigaDOS, and give a program the most control over devices. For example, when accessing the serial device using the Amiga functions, a program has control over the device's baud rate, etc., which it does not have when accessing the serial device using the ANSI-compatible or AmigaDOS functions.

The ANSI-compatible functions supplied with Aztec C support console I/O differently on other systems than they do on the Amiga. On other systems, there is one console device and a program can issue calls to the `ioctl()` function to define how console I/O is to be performed. On the Amiga, `ioctl()` is only supported for setting RAW mode. See the `ioctl()` command in the **Library Functions** chapter of this manual for more information.

Aborting a Program from the Console

To abort a program, the operator types ^C which `Chk_Abort()` detects. `Chk_Abort()` calls a routine called `_abort()` which prints a ^C and then calls `exit()`. This routine can be replaced by the program so that the program can do any cleaning up that is necessary before exiting.

When called, **Chk_Abort()** aborts the program if (1) ^C is typed and (2) the global long **Enable_Abort()** is nonzero (which it is by default). If either of these conditions is not satisfied, **Chk_Abort()** simply returns.

Therefore, to disable the ability of the operator to abort a program by typing ^C, a program must set **Enable_Abort()** to 0. **Chk_Abort()** is called in all the low level I/O routines such as **open()**, **read()**, **write()**, and **lseek()**.

Overview of Aztec Standard I/O

Aztec C Standard I/O Functions

The standard I/O functions are used by programs to access files and devices. Aztec C functions are fully compatible with the ANSI standard.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, they handle the blocking and unblocking of file data. Therefore, your program can concentrate on its own concerns.

Opening Files and Devices

Before a program can access a file or device, it must be "opened," and when processing on a file or device is done, it must be "closed."

An open device or file is called a **STREAM** and has associated with it a pointer, called a **FILE POINTER**, to a structure of type **FILE**. This identifies the file or device when standard I/O functions are called to access it.

It contains such information as the file position, pointer to an associated buffer, an error indicator, and an end-of-file indicator.

There are two ways for a file or device to be opened for standard I/O: first, the program can explicitly open it, by calling one of the functions **fopen()**, **freopen()**, or **fdopen()**. In this case, the **open()** function returns the file pointer associated with the file or device. **fopen()** opens the file or device. **freopen()** reopens an open stream to another file or device; it is mainly used to change the file or device associated with one of the logical devices (standard output, standard input, or standard error.) **fdopen()** opens for standard I/O a file or device already opened for unbuffered I/O.

Alternatively, the file or device can be automatically opened as one of the logical devices (standard input, standard output, or standard error.) In this case, the file pointer is `stdin`, `stdout`, or `stderr`, respectively. These symbols are defined in the header file `stdio.h`.

Closing Streams

A file or device opened for standard I/O can be closed in two ways:

- The program can explicitly close it by calling the function `fclose()`.
- When the program terminates, either by falling off the end of the function `main()`, or by calling the function `exit()`, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: Data written to files using the standard I/O functions is buffered in memory, and a buffer is not written to the file until it is full or the file is closed. Most likely, when a program finishes writing to a file, the file buffer will be partially full, with this information not having been written to the file. If a program calls `fclose()`, this function will write the partially filled buffer to the file and return an error code if this could not be done. If the program lets the system automatically close the file, the program will not know if an error occurred on this last write operation.

Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the current position of the file and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero if opened for read or write access, and to its end if opened for append.

Random I/O

The function `fseek()` allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

Buffering

When the standard I/O functions are used to access a file, the I/O is buffered. Either a user-specified or dynamically-allocated buffer can be used.

Your program specifies a buffer to be used for a file by calling the function `setbuf()` after the file has been opened but before the first I/O request to it has been made.

If, when the first I/O request is made to a file, you have not specified the buffer to be used for the file, the system will automatically allocate, by calling `malloc()`, a buffer for it. When the file is closed its buffer will be freed, by calling `free()`.

Dynamically allocated buffers are obtained from a particular region of memory (the heap), whether requested by the standard I/O functions or by your program.

The default size of an I/O buffer is given by the manifest constant `BUFSIZ` in `stdio.h`.

By default, output to the console using standard I/O functions is unbuffered; all other device I/O using the standard I/O functions is buffered. Console input buffering can be disabled using the `setbuf()` function.

Errors

There are three fields that may be set when an exceptional condition occurs during stream I/O. Two of the fields are unique to each stream (i.e., each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file (EOF) is detected on input from the stream; the other is set if an error occurs during I/O to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the `clearerr()` function for the stream. The only exception to the last statement is that when called, `fseek()` will reset the EOF flag for a stream. A program can check the status of the EOF and error flags for a stream by calling the functions `feof()` and `ferror()`, respectively.

The other field which may be set is the global integer `errno`. By convention, a system function which returns an error status as its value can also set a code in `errno` which more fully defines the error.

If an error occurs when a stream is being accessed, a standard I/O function returns EOF (-1) as its value, after setting a code in `errno` and setting the stream's error flag.

If EOF is reached on an input stream, a standard I/O function returns EOF after setting the stream's EOF flag.

There are two techniques a program can use for detecting errors during stream I/O. First, the program can check the result of each I/O call. Second, the program can issue I/O calls and only periodically check for errors (for example, check only after all I/O is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling `ferror()` is more efficient. When characters are written to a file using the standard I/O functions they are placed in a buffer, which is not written to disk until it is full. If the buffer is not full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient.

Once a file opened for standard I/O is closed, `ferror()` cannot be used to determine if an error has occurred while writing to it. Thus `ferror()` should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by `fclose()`, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, its standard I/O buffer will probably be partly full. This buffer will be written to the file when the file is closed, and `fclose()` will return an error status if this final write operation fails.

THE STANDARD I/O FUNCTIONS

The standard I/O functions can be grouped into two sets: those that can access only the logical devices (standard input, standard output, and standard error); and all the rest. **Functions marked obsolete will not be supported in future releases. You will also need to refer to release 3.6 documentation for detailed specifications for some of these functions.**

Here are the standard I/O functions that can only access **stdin**, **stdout**, and **stderr**. These are all ASCII functions; i.e., they expect to deal with text characters only.

getchar()	Get an ASCII character from stdin
gets()	Get a line of ASCII characters from stdin
printf()	Format data and send it to stdout
*puterr()	Send a character to stderr - obsolete
putchar()	Send a character to stdout
puts()	Send a character string to stdout
scanf()	Get a line from stdin and convert it
vprintf()	Format data using a variable argument list and send it to stdout .

Here are the rest of the standard I/O functions. Those preceded by an asterisk (*) are non-ANSI functions:

clearerr()	Clear EOF and error flag for stream
fclose()	Close an open stream
*fdopen()	Open as a stream a file or device already open for unbuffered I/O
feof()	Check for EOF on a stream
ferror()	Check for error on a stream
fflush()	Write stream's buffer
fgetc()	Get the next character from an input stream
fgetpos()	Get information on file position
fgets()	Get a line of ASCII characters
*fileno()	Get file descriptor associated with stream
fopen()	Open a file or device
fprintf()	Format data and write it to a stream
fputc()	Write a character to an output stream
fputs()	Send a string of ASCII characters to a stream
fread()	Read data from a specified stream.
freopen()	Open an open stream to another file or device
fscanf()	Get data and convert it

fseek()	Set current position within a file
fsetpos()	Set file position
ftell()	Get current position
fwrite()	Write data to a stream
getc()	Macro for fgetc()
*getw()	Get two characters from stream - obsolete
perror	Map errno to a descriptive string
putc	Macro for fputc()
*putw	Send two characters to a stream - obsolete
remove	Deletes a file
rename	Changes the name by which a file is accessed
rewind	Set file position to
setbuf	Specify buffer and buffering
setvbuf	Specify buffer, buffering, and buffer size
tmpfile	Creates a temporary binary file
tmpnam	Generates a unique name for a file
ungetc	Push character back into stream
vfprintf	Format data using a variable argument list and write to a stream

AMIGA STANDARD I/O

Buffering

On the Amiga, the size of a buffer used for standard I/O is 1024 bytes.

Overview of Aztec C and Amiga Unbuffered I/O

The Aztec C unbuffered I/O functions are used to access files and devices. They are similar to their UNIX counterparts with the addition that a file may be opened as either text or binary. The Amiga unbuffered I/O functions are described at the end of this section.

A program using these functions communicates directly with files and devices; data does not pass through system buffers. Some unbuffered I/O, however, is buffered. When data is transferred to or from a file in blocks

smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is buffered, unless specific actions are taken by your program.

Programs which use the unbuffered I/O functions to access files generally handle the blocking and unblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and unblocking can use the standard I/O functions.

Here are the unbuffered I/O functions:

access()	Determine if a file can be accessed in a specified manner.
close()	Conclude the I/O on an open file or device
creat()	Create a file and open it
ioctl()	Change console I/O mode
isatty()	Is an open file or device the console?
lseek()	Change the current position of an open file
open()	Prepare a file or device for unbuffered I/O
read()	Read data from an open file or device
rename()	Rename a file
unlink()	Delete a file
write()	Write data to an open file or device

The header file **fcntl.h** is designed to be used in conjunction with the unbuffered I/O routines. It includes function declarations, macros to be used with these functions, and the **_dev** structure declaration.

Before a program can access a file or device, it must be opened, and when processing on it is done, it must be closed.

An open file or device has an integer known as a **FILE DESCRIPTOR** associated with it; this identifies the file or device when it is accessed.

There are two ways for a file or device to open for unbuffered I/O. First, you can explicitly open it, by calling the function **open()**. In this case, **open()** returns the file descriptor to be used when accessing the file or device.

Alternatively, you can automatically open the file or device as one of the logical devices (standard input, standard output, or standard error.) In this case, the file descriptor is the integer value 0, 1, or 2, respectively.

An open file or device is closed by calling the function `close()`. When a program ends, any devices or files still opened for unbuffered I/O will be closed.

If an error occurs during an unbuffered I/O operation, the function returns -1 as its value and sets a code in the global integer `errno`.

FILE I/O

Programs call the functions `read()` and `write()` to access a file; the transfer begins at the current position of the file and proceeds until the number of characters specified by the program has been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random access to the file. For sequential access, a program simply issues consecutive I/O requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function `lseek()` provides random access to a file by setting the current position to a specified character location.

`lseek()` allows the current position of a file to be set relative to the end of a file. For systems which do not keep track of the last character written to a file, such positioning cannot always be correctly done.

`open()` provides a mode, `O_APPEND`, which causes the file being opened to be positioned at its end. As with `lseek()`, the positioning may not be correct for systems which do not keep track of the last character written to a file. `open()` also provides for opening files as text by using the `O_TEXT` mode. The default mode is binary.

DEVICE I/O

Unbuffered I/O to the Console

There are several options available when accessing the console. Here we want to briefly discuss the line- or character-modes of console I/O as they relate to the unbuffered I/O functions.

Console input can be either line- or character-oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called unbuffered I/O functions.

With character-oriented input, characters are read and returned to the program when they are typed; no buffering of console input occurs.

Unbuffered I/O to Nonconsole Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

Overview of Aztec C and Amiga Console I/O

A program has control over several options relating to console I/O. The primary option supported by the Amiga is the RAW mode, described below.

Character-Oriented Input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console I/O features available for a particular system.

The type of character-oriented console input which the Amiga supports is type RAW. In RAW mode, a program does not have control over the other console options.

Writing System-Independent Programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console.

The standard I/O functions all read one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard I/O functions.

Some systems require a program that wants to set console options to first call `ioctl()`, to fetch the current console options, then modify them as desired, and finally call `ioctl()` to reset the new console options. The systems that do not require this do not care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console I/O options other than the default should fetch the current console options before modifying them.

Using `ioctl`

A program selects console I/O modes using the function `ioctl()`. This has the form:

```
#include sgtty.h
ioctl(fd, code, arg)
struct sgtyb *arg;
```

The header file `sgtty.h` defines symbolic values for the code parameter (which tells `ioctl` what to do) and the structure `sgtyb`.

The parameter `fd` is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the `ioctl` call applies. Here, `ioctl` always applies to the console.

The parameter `code` defines the action to be performed by `ioctl()`. It can have these values:

TIOCGTP	Fetch the console parameters and store them in the structure pointed at by <i>arg</i> .
TIOCSPT	Set the console parameters according to the structure pointed at by <i>arg</i> .

The argument **arg** points to a structure named **sgttyb** that contains the following field:

```
int sg_flags;
```

To set console options, a program should fetch the current state of the fields, using **ioctl()**'s **TIOCGTP** option. Then it should modify the fields to the appropriate values and call **ioctl()** again, using the **ioctl()** **TIOCSPT** option.

THE **sgtty** FIELD

sg_flags contains the UNIX-compatible flag **RAW**, which sets **RAW** mode and turns off other options. By default, **RAW** is disabled.

On some systems, other flags are contained in **sg_flags**.

More than one flag can be specified in a single call to **ioctl()**; the values are simply **ORED** together. If the **RAW** option is selected, none of the other options have any effect.

When the console I/O options are set and **RAW** is reset, the console is set in line-oriented input mode.

EXAMPLE

Console Input - **RAW** Mode

In this example, a program opens the console for standard I/O, sets the console in **RAW** mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator are not displayed unless the program itself displays them. The input request will not terminate until a character is received.

This example assumes that the console is named `con`; on systems for which this is not the case, just substitute the appropriate name.

```
include <stdio.h>
#include <sgtty.h>
main()
{
    int c;
    FILE *fp;
    struct sgttyb stty;
    if ((fp = fopen("CON:x/y/width/
        height/", "r")) == NULL){
        printf("can't open the console\n");
        exit();
    }
    ioctl(fileno(fp), TIOCGETP, &stty);
    stty.sg_flags |= RAW;
    ioctl(fileno(fp), TIOCSETP, &stty);
    for (;;) {
        c = getc(fp);
        . . .
    }
}
```

Overview of Aztec C and Amiga Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the heap. They are:

malloc()	Allocate a buffer
calloc()	Allocate a buffer and initialize it to zeroes
realloc()	Allocate more space to a previously allocated buffer
free()	Release an allocated buffer for reuse

These functions are described in detail in the **Library Functions** chapter of this manual.

In addition, on some systems the UNIX-compatible functions `sbrk()` and `brk()` provide a more elementary means to allocate heap space. The malloc-type functions call `sbrk()` to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported.

Dynamic Allocation of Standard I/O Buffers

Buffers used for standard I/O are dynamically allocated from the heap unless specific actions are taken by your program. Standard I/O calls to dynamically allocate and deallocate buffers can be interspersed with those of your program.

Programs that perform standard I/O and have absolute control of the heap can explicitly define the buffers to be used by a standard I/O stream.

Overview of Dynamic Buffer Allocation

The Amiga function `lmalloc()` is a non-UNIX memory allocation function. It is obsolete. It is supported in this release for compatibility. There is no detailed information for `lmalloc()` in this document.

Note: For more information about specific functions, see the **Library Functions** chapter. For more information about the heap, see the "Program Organization" section of the **Technical Information** chapter of your *Aztec C Reference Manual*.

Overview of Aztec C and Amiga Error Processing

This section discusses error processing which relates to the global integer `errno`. This variable is modified by the standard I/O, unbuffered I/O, and floating point math (e.g., `sin()`, `sqrt()`) functions as part of their error processing.

When a standard I/O, unbuffered I/O, or math function detects an error, it sets a code in `errno` which describes the error. If no error occurs, the math functions do not modify `errno`. If no error occurs, the I/O functions may or may not modify `errno`.

Also, when an error occurs,

- A standard I/O function returns -1 and sets an error flag for the stream on which the error occurred;
- An unbuffered I/O function returns -1;
- A math function returns a value dependent on the type of error that occurred and sets **errno**. For example, an overflow would return **HUGE_VAL**.

When performing math calculations, a program can check **errno** for errors as each function is called. Alternatively, since **errno** is modified only when an error occurs, **errno** can be checked only after a sequence of operations; if it is nonzero, an error has occurred at some point in the sequence. This latter technique can only be used when no I/O operations occur during the sequence of math function calls.

Since **errno** may be modified by an I/O function even if an error did not occur, a program cannot perform a sequence of I/O operations and then check **errno** afterwards to detect an error. Programs performing unbuffered I/O must check the result of each I/O call for an error.

Programs performing standard I/O operations cannot, following a sequence of standard I/O calls, check **errno** to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard I/O operations on a stream and then check the stream's error flag.

The following table lists the system-independent values that may be placed in **errno**. These symbolic values are defined in the file **errno.h**. Other system-dependent values may also be set in **errno** following an I/O operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

<i>Error code</i>	<i>Meaning</i>
ENOENT	File or directory does not exist
E2BIG	Argument list too long
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat() request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	ioctl() attempted on non-console device
EACCES	Invalid access request, permission denied
EIO	I/O error (usually physical)
ENOSPC	No space left on device
ERANGE	Math function result too large
EDOM	Domain of argument to math function invalid
ENOEXEC	fexec() format error
EROFS	Read-only file system
EXDEV	Cross-device rename
EAGAIN	Nothing to read

Overview of ANSI Header Files

Header files may be included in any order. They may be included more than once in a given scope, without any adverse effects. The header files include:

assert.h

Includes the **assert()** macro which may be used to display the file name and line number for a call that fails.

ctype.h

Includes several function declarations and macros useful for testing and mapping characters.

errno.h

Includes macros for the reporting of error conditions. See the section **Overview of Aztec C and Amiga Error Processing**.

fcntl.h

Includes function declarations and macros for use with unbuffered I/O.

float.h

Includes macros that define various floating point sizes and limits.

limits.h

Includes macros that define the numerical limits for various ordinal types.

locale.h

Includes macros and function declarations supporting the "C" locale necessary for C translation.

math.h

Includes declarations for several mathematical functions that take double-precision arguments and return double-precision values.

setjmp.h

Includes the **setjmp()** macro for use with the **longjmp()** function.

signal.h

Includes function declarations and macros for handling various signals.

stdarg.h

Includes macros for advancing through a variable length argument list.

stddef.h

Includes type definitions for pointer subtraction, result of **sizeof** operator (which is now 32 bits), and largest available character set.

stdio.h

Includes types, macros, and function declarations for buffered I/O.

stdlib.h

Includes types and function declarations for string conversion, memory management, and other general utilities.

string.h

Includes function declarations for manipulating arrays of character type.

time.h

Includes types and function declarations for obtaining the current date and time as well as manipulating it and other time structures.

Chapter 3 - Library Functions

Introduction

This chapter contains two categories of library functions:

- System-independent functions that are common to all Aztec C packages.
- Machine-dependent functions that work only on the Amiga machines.

Note: The Amiga WorkBench routines are also discussed in their own chapter.

To help you access the information more easily and efficiently:

- Each function description is listed alphabetically
- Each function description begins on a new page.

This introduction includes a list of all included library functions as well as an explanation of the format that the function descriptions feature.

The individual function descriptions begin on page 3-9.

FUNCTION LIST

The functions described in this chapter include:

- _abort()** - abort task
- abs()** - return the absolute value of a signed integer
- access()** - determine accessibility of a file or directory
- acos()** - return the arc cosine of a double value
- asctime()** - translate a time value into an ASCII string
- asin()** - return the arc sine of a double value
- assert()** - verify program assertion
- atan()** - return the arc tangent of a double value
- atan2()** - return the arc tangent of a double value y/x
- atexit()** - indicate a function to be called at program termination
- atof()** - convert ASCII string to a double number
- atoi()** - convert an ASCII string to a signed integer
- atol()** - convert an ASCII string to a signed long value
- calloc()** - allocate space for an array of objects from system memory
- ceil()** - compute the smallest integer not less than a specified number
- clearerr()** - clear end of file and error conditions in a stream
- _cli_parse()** - parse command line
- clock()** - determine time intervals
- close()** - close a device or file
- cos()** - return the cosine of a double value
- cosh()** - return the hyperbolic cosine of a double value
- cotan()** - returns the cotangent of a double value
- creat()** - create a new file
- ctime()** - convert a time value to an ASCII string
- difftime()** - compute the difference between two times
- div()** - compute the quotient and remainder of the division of two integers
- dos_packet()** - output packet
- exec1(), execv(), execlp(), execvp()** - load and start another program
- exit(), _exit()** - terminate calling program
- exp()** - compute the exponential function e^x

fabs() - return the absolute value of a given number
fclose() - close a buffered I/O stream
fdopen() - open a file or device previously opened for buffered I/O for standard I/O
feof() - test for end-of-file in a standard I/O stream
ferror() - test for an error in a standard I/O stream
fexec1(), **fexecv()** - load and start another program
fflush() - flush an I/O stream
fgetc() - return the next available character from a standard I/O stream
fgetpos() - save the current file position for a stream
gets() - get a string of characters from a stream
fileno() - return file descriptor associated with a stream
floor() - compute the largest integer not greater than a specified number.
fmod() - return the remainder of the double value x/y
fopen() - open a specified file or device for standard I/O access
format() - write formatted data using a user-defined output function
fprintf() - write formatted data to an I/O stream
fputc() - write a character to an I/O stream
fputs() - write a string to an I/O stream
fread() - read from a specified standard I/O stream
free() - deallocate a memory block
freeseq() - segment unload function
freopen() - reopen a stream with a new device
frexp() - decompose a floating point number
fscanf() - perform formatted input conversion on a specified stream
fseek() - reposition current location within a stream
fsetpos() - set the correct file position for a stream
ftell() - return the current file position within a standard I/O stream
ftoa() - convert floating point number to an ASCII string
fwrite() - write to a specified standard I/O stream
geta4() - set up a4 register
getc() - return the next available character from a standard I/O stream
getchar() - return the next available character from **stdin**
getenv() - get value of environment variable

gets() - get a string of characters from **stdin**
gmtime() - convert a calendar time into coordinated universal time
index() - find the first occurrence of a character in a string
int_end() - restore machine state from within a C interrupt handler
int_start() - set up a function as an interrupt handler
ioctl() - device I/O utility
labs() - return the absolute value of a signed long
ldexp() - build real numbers
ldiv() - compute quotient and remainder of the division of two longs
localtime() - convert a time value relative to local time
log() - compute the natural logarithm of a number
log10() - compute the logarithm of a number to base 10
longjmp() - execute a non-local **goto**
lseek() - change current position within file
malloc() - allocate a block of system memory
memchr() - find a character within an object
memcmp() - compare two blocks of memory *n* bytes long
memcpy() - copy a block of bytes from one object to another
memmove() - copy a block of memory
memset() - set a block of memory to a specified value
mktemp() - make a unique file name
mktime() - convert a time value between formats
modf() - break a floating point value into integral and fractional parts
movmem() - copy a block of memory
open() - open device or file for unbuffered I/O
perror() - print a system error message
pow() - compute *x* to the *y*th power
printf() - formatted output function
putc() - write a character to an I/O stream
putchar() - write a character to the **stdout** stream
puts() - write a string to **stdout**
qsort() - sort an array of records in memory
ran() - generate floating point random numbers
rand() - return a pseudo-random integer
read() - read from a device or file using unbuffered I/O

realloc() - re-allocate an already existing memory block to a different size

remove() - delete a file

rename() - rename a disk file

rewind() - reposition a stream's position indicator to the beginning of the file

rindex() - find the last occurrence of a character in a string

sbrk() - memory allocation function

scanf() - perform formatted input conversion on the **stdin** stream

scdir() - return the name of the next file matching pattern

segload() - load a program segment

setbuf() - associate an I/O stream with a specific buffer

setenv() - set environment variables

setjmp() - set up for a non-local **goto**

setvbuf() - associate an I/O stream with a specific buffer

signal() - define how to handle a signal

sin() - return the sine of a double value

sinh() - return the hyperbolic sine of a double value

sprintf() - write formatted data to a buffer

sqrt() - compute the non-negative square root of a value.

srand() - set the random number seed for **rand()**

rand() - initialize the seed value used by **rand()**

sscanf() - perform formatted input conversion on a buffer

stat() - return file information

_stkchk() - check for stack overflow

_stkover() - report stack overflow

strcat() - concatenate two strings together

strchr() - search for the first occurrence of a character in a string

strcmp() - compare two strings

strcoll() - compare two strings using the current locale

strcpy() - copy one string to another

strcspn() - return the index of the first character in a string *str1* which is contained in a string *str2*

strlen() - return the length of a string

strncat() - concatenate two strings together, up to *max* characters
strncmp() - compare two strings, up to *max* characters.
strncpy() - copy up to *max* characters from one string to another
strpbrk() - return the first occurrence in a string *str1* of a character in
a string *str2*.
strrchr() - search for the last occurrence of a character in a string
strspn() - return the index of the first character in a string *str1* which
is not contained in a string *str2*
strstr() - return a pointer of the first occurrence of a string *str2* within
a string *str1*
strtod() - convert a string to a double
strtok() - tokenize a string
strxfrm() - transform a string to match the current locale
tan() - return the tangent of a double value
tanh() - return the hyperbolic tangent of a double value
time() - return the time of day
tmpfile() - create a temporary file
tmpnam() - create a name for a temporary file
tolower() - convert a character to lowercase
toupper() - convert a character to uppercase
ungetc() - push a character back into input stream
unlink() - erase file
va_arg(), va_end(), va_start() - variable argument access
vfprintf() - write formatted ASCII data to a stream
vprintf() - write formatted ASCII data to **stdout**
vsprintf() - write formatted ASCII data to a buffer
_wb_parse() - open standard I/O window
write() - write to a file or device using unbuffered I/O

DESCRIPTION FORMAT

The description format is as shown on the following pages.

TYPE	Lists the name of the function, one or more letters in parentheses which specify the libraries that contain the function, and the function category. The letters which may appear in parentheses and their corresponding libraries are: <table><tr><td>C</td><td>c.lib, c16.lib, cl.lib, cl16.lib</td></tr><tr><td>M</td><td>m.lib, m8.lib, mf.lib, ma.lib</td></tr><tr><td>S</td><td>s.lib</td></tr></table> At the right margin, in italics, is the standard to which the function adheres, (i.e., ANSI, UNIX, etc.)	C	c.lib, c16.lib, cl.lib, cl16.lib	M	m.lib, m8.lib, mf.lib, ma.lib	S	s.lib
C	c.lib, c16.lib, cl.lib, cl16.lib						
M	m.lib, m8.lib, mf.lib, ma.lib						
S	s.lib						
NAME	Lists the name of the function and a brief description of its use.						
SYNOPSIS	Indicates the types of arguments that the functions require, and the values it returns. For example, the function <code>atof</code> converts character strings into double precision numbers. It is listed in the synopsis as <pre>double atof(const char *cp);</pre> This means that <code>atof()</code> returns a value of type <code>double</code> and requires as an argument a pointer to a character string. Since <code>atof()</code> returns a noninteger value, it must be declared prior to the use of the function: <pre>double atof(const char *cp);</pre> Unless otherwise noted, the function's declaration is contained in the header file specified for that function. The notation <pre>#include <stdlib.h></pre> at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling that function.						
DESCRIPTION	Describes the function.						

SEE ALSO	Lists other relevant functions. You should also refer to the Library Overview chapter for further information about many of the functions discussed in this chapter.
DIAGNOSTICS	Describes the error codes that the function may return. The Errors Section in the Library Overview chapter presents an overview of error processing.
EXAMPLES	Gives examples on use of the function.

TYPE	<code>_abort</code> (C) - Miscellaneous	<i>Amiga</i>
NAME	<code>_abort</code> - abort task	
SYNOPSIS	<pre>void abort(void);</pre>	
DESCRIPTION	<code>_abort()</code> is a function that is called by the function <code>Chk_Abort()</code> if the flag Enable-Abort is nonzero and the ^C key is entered by the user (or if the ^C signal is sent to the task in general.) The library version of <code>_abort()</code> displays ^C on the screen and calls <code>exit(1)</code>. You may provide your own <code>_abort()</code> routine to take special action if ^C is pressed. For example, the compiler, assembler, and linker remove any temporary files when ^C is typed. <code>_abort()</code> calls a cleanup function that deletes the files, prints a message, and then calls <code>exit()</code>.	

TYPE	abs (C) - Integer Math	ANSI
NAME	abs - return the absolute value of a signed integer	
SYNOPSIS	<pre>#include <stdlib.h> int abs (int x);</pre>	
DESCRIPTION	abs() computes and returns the absolute value of the signed integer <i>x</i> . The largest possible negative integer is returned as itself, since the largest negative integer cannot be represented positively in the size of an integer.	
SEE ALSO	labs(), fabs()	

TYPE	access (C) - Standard I/O	Aztec										
NAME	access - determine accessibility of a file or directory											
SYNOPSIS	#include <fcntl.h> int access (char *filename, int mode);											
DESCRIPTION	access() determines whether a file or directory can be accessed in a specified manner. The parameter <i>filename</i> points to the name of the file or directory; this name optionally contains the drive and path of the directories that must be passed through to get to the file or directory. If the drive component is not specified, the file or directory is assumed to reside on the default drive. If the path component is not specified, the file or directory is assumed to reside in the current directory on the specified drive. <i>mode</i> is an int that specifies the type of access desired: <table><tr><th>Mode</th><th>Meaning</th></tr><tr><td>4</td><td>read</td></tr><tr><td>2</td><td>write</td></tr><tr><td>1</td><td>execute (if a file) or search (if a directory)</td></tr><tr><td>0</td><td>existence of the file or directory.</td></tr></table> If the existence of the file or directory is being checked (i.e., <i>mode</i> = 0), access() returns 0 if the file exists and -1 if it does not. In the latter case, access() also sets the symbolic value ENOENT in the global integer <i>errno</i>. When access() is called to determine if a file can be accessed in a certain way (i.e., <i>mode</i> is not 0), access() returns 0 if the file can be accessed in the desired manner; otherwise, it returns -1 and sets a code in the global integer <i>errno</i> that defines why the access is not permitted.		Mode	Meaning	4	read	2	write	1	execute (if a file) or search (if a directory)	0	existence of the file or directory.
Mode	Meaning											
4	read											
2	write											
1	execute (if a file) or search (if a directory)											
0	existence of the file or directory.											

When read or write privileges are requested on a directory, `access()` reports on whether files can be read or written to within the directory, not whether or not the directory itself may be read or written to.

The symbolic values that `access()` may set in `errno` when it is called with a non-zero *mode* parameter are:

<i>errno</i>	<i>Meaning</i>
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	The file or directory does not exist.
EACCES	File or directory cannot be accessed in the desired manner

TYPE	acos (M) - Float Math	ANSI
NAME	acos - return the arc cosine of a double value	
SYNOPSIS	<pre>#include <math.h> double acos (double x);</pre>	
DESCRIPTION	acos() returns the arc cosine of x. x should be specified in radians. If x is outside of the range -1 to 1, then acos() will return 0 and set errno equal to EDOM.	

TYPE	asctime (C) - Time	ANSI
NAME	asctime - translate a time value into an ASCII string	
SYNOPSIS	<pre>#include <time.h> char *asctime (const struct tm *timeptr);</pre>	
DESCRIPTION	asctime() creates an ASCII text string corresponding to the time contained in <i>timeptr</i>. The general format of the resulting string is: <pre>Mon Apr 9 08:29:00 1968\n\o</pre> A pointer to the text string is returned by asctime.	
SEE ALSO	clock(), ctime(), difftime(), localtime(), time()	

TYPE	asin (M) - Float Math	<i>ANSI</i>
NAME	asin - return the arc sine of a double value	
SYNOPSIS	<pre>#include <math.h> double asin(double x);</pre>	
DESCRIPTION	asin() returns the arc sine of x. x should be specified in radians. If x is outside of the range -1 to 1, then asin() will return 0 and set errno equal to EDOM.	

TYPE	assert (Macro) - Miscellaneous	ANSI
NAME	assert - verify program assertion	
SYNOPSIS	<pre>#include <assert.h> void assert (int <i>expression</i>);</pre>	
DESCRIPTION	The assert() macro is useful for putting diagnostic messages in a program. assert() determines whether <i>expression</i> is true or false. If <i>expression</i> evaluates to false, the message: <pre>Assertion failed: expr, file ffff, line lnnn</pre> is printed to stdout, where ffff is the name of the source file and nnn is the line number where the assertion failed. To prevent assertion statements from being compiled in a program, compile the program with the option -dNDEBUG, or place the statement #define NDEBUG ahead of the statement #include assert.h.	

TYPE	atan (M) - Float Math	<i>ANSI</i>
NAME	atan - return the arc tangent of a double value	
SYNOPSIS	<pre>#include <math.h> double atan (double x);</pre>	
DESCRIPTION	atan() returns the arc tangent of <i>x</i> . <i>x</i> should be specified in radians.	

TYPE	atan2 (M) - Float Math	ANSI
NAME	atan2 - return the arc tangent of a double value y/x	
SYNOPSIS	<pre>#include <math.h> double atan2 (double y, double x);</pre>	
DESCRIPTION	atan2() returns the arc tangent of the ratio of the input values (y/x) in the range of $-\pi$ to π. atan2() will use the signs of the input values to determine the correct quadrant of the return value. If both x and y are 0, atan2() returns 0 and sets errno equal to EDOM.	

TYPE	atexit (C) - Miscellaneous	ANSI
NAME	atexit - indicate a function to be called at program termination	
SYNOPSIS	<pre>#include <stdlib.h> int atexit (void (*func) (void));</pre>	
DESCRIPTION	atexit() is used to indicate that the function <i>func</i> should be called at normal program termination. When a program returns from main(), or the exit() function is called, atexit() insures that <i>func</i> is called. Up to 32 functions may be indicated as exit() functions with atexit(). atexit() will return a non-zero value if the function cannot be registered as an exit() function; otherwise, it returns 0.	
SEE ALSO	exit()	

TYPE	atof (M) - Conversion	ANSI
NAME	atof - convert ASCII string to a double number	
SYNOPSIS	<pre>#include <stdlib.h> double atof(const char *cp);</pre>	
DESCRIPTION	atof() converts a string of text characters pointed at by the argument <i>cp</i> to a double. The string may contain leading blanks and tabs, which it skips, followed by an optional sign (+ or -), then a number containing an optional decimal point (.), then an optional "E" or "e", followed by an optionally signed integer to denote scientific notation. Any characters beyond this are ignored.	
EXAMPLE	<pre>#include <stdlib.h> main() { double val; char *cp; cp = "10.6789"; val = atof (cp); printf (" val = %e\n", val); }</pre>	
SEE ALSO	atoi(), atol(), ftoa()	

TYPE	atoi(C) - Conversion	ANSI
NAME	atoi - convert an ASCII string to a signed integer	
SYNOPSIS	<pre>#include <stdlib.h> int atoi (const char *cp);</pre>	
DESCRIPTION	atoi() converts a string of text characters pointed to by the argument <i>cp</i> into a signed integer value, which it returns. The format of the string pointed at by <i>cp</i> should contain three components: optional leading blanks or tabs, followed by an option "+" or "-", followed by an integer. Any numbers following the integer will be ignored, although the string should be null-delimited.	
EXAMPLE	<pre>#include <stdlib.h> main() { int i; char *cp = " 567"; i = atoi (cp); printf ("I = %d\n", i); }</pre>	
SEE ALSO	atof(), atol(), ftoa()	

TYPE	atol (C) - Conversion	ANSI
NAME	atol - convert an ASCII string to a signed long value	
SYNOPSIS	<pre>#include <stdlib.h> long atol (const char *cp);</pre>	
DESCRIPTION	atol() converts the string of characters pointed to by the argument <i>cp</i> to a signed long, which is then returned by atol(). The string may contain optional leading blanks, followed by an optional "+" or "-", followed by a string of digits. Anything beyond the string of digits is ignored. The string should also be null delimited.	
EXAMPLE	<pre>#include <stdio.h> #include <stdlib.h> main() { long val; char *cp = " 79832"; val = atol (cp); printf ("val = %ld\n", val); }</pre>	
SEE ALSO	atof(), atoi(), ftoa()	

TYPE	calloc (C) - Memory Allocation	ANSI
NAME	calloc - allocate space for an array of objects from system memory	
SYNOPSIS	<pre>#include <stdlib.h> void *calloc (size_t <i>nmemb</i>, size_t <i>size</i>);</pre>	
DESCRIPTION	calloc() allocates system memory for an array of <i>nmemb</i> objects, each <i>size</i> bytes long, in a manner similar to the malloc() function. The total size of the block will be <i>size</i> * <i>nmemb</i> bytes long, and each byte within the block is initialized to 0. If calloc() is successful, it returns a pointer to the allocated block.	
SEE ALSO	malloc(), realloc(), free(), lmalloc()	
DIAGNOSTICS	If calloc() cannot allocate a block large enough to hold <i>nmemb</i> elements each <i>size</i> bytes long, it will return a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	ceil (M) - Float Math	ANSI
NAME	ceil - compute the smallest integer not less than a specified number	
SYNOPSIS	<pre>#include <math.h> double ceil (double x);</pre>	
DESCRIPTION	ceil() will return the smallest integral number not less than its input, <i>x</i> . The return value is expressed as a double . For example, ceil() will return 6.0 for an input of 5.3, and -5.0 for an input of -5.3.	
SEE ALSO	fabs(), floor()	

TYPE	clearerr (C) - Standard I/O	ANSI
NAME	clearerr - clear end of file and error conditions in a stream	
SYNOPSIS	<pre>#include <stdio.h> void clearerr(FILE *stream);</pre>	
DESCRIPTION	clearerr() clears both the end-of-file and error condition codes associated with the specified <i>stream</i> . If an error or end-of-file condition occurs on a stream and clearerr() is not called, the error condition will remain set until the stream is closed.	
SEE ALSO	feof(), ferror(), perror()	

TYPE	<code>_cli_parse (C) - Amiga</code> <i>Amiga</i>
NAME	<code>_cli_parse</code> - parse command line
SYNOPSIS	<pre>extern int _argc, _arg_len; extern char** _argv, *_arg_line; void _cli_parse (struct process *pp, long char *aptr)</pre>
DESCRIPTION	<code>_cli_parse()</code> is a function contained within the Aztec library which is called upon starting of a program from the CLI. Its purpose is to parse the command line string passed to the program from the CLI command line and convert it into <code>argc-argv</code> format. <code>_cli_parse()</code> is called by the routine <code>_main</code> with the following arguments: • <code>pp</code> points to the program's Process structure • <code>aptr</code> points to the command string • <code>alen</code> is the length of the command string. <code>_cli_parse</code> parses the command string, allocates memory for the parsed information, and sets up pointers to the parsed information in the global variables <code>_argc</code>, <code>_arg_len</code>, <code>argv</code>, and <code>_arg_line</code>. These variables are used by <code>_main</code> when calling <code>main</code>. This allocated memory is freed by <code>_exit()</code>. <code>_cli_parse</code> is supplied as a separate module. Programs that wish to do custom argument parsing may supply their own routine that overrides the one from the library. It is also possible to produce smaller programs by replacing this function with a null function.

TYPE	clock (C) - Time	<i>ANSI</i>
NAME	clock - determine time intervals	
SYNOPSIS	<pre>#include <time.h> clock_t clock (void);</pre>	
DESCRIPTION	clock() is used to determine the time interval between two events. The 0 value returned by clock() should be divided by the macro CLF_TCK to determine the time in seconds.	

TYPE	close (C) - UNIX I/O	<i>Aztec /UNIX</i>
NAME	close - close a device or file	
SYNOPSIS	<pre>#include <fcntl.h> int close (int <i>fd</i>);</pre>	
DESCRIPTION	close() closes a device or disk file which has been opened for unbuffered I/O. The parameter <i>fd</i> is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling open() or creat(), <i>fd</i> is the file descriptor returned by open() or creat(). close returns 0 as its value if successful.	
DIAGNOSTICS	If close() fails, it returns -1 and sets an error code in the global integer errno.	

TYPE	cos (M) - Float Math	ANSI
NAME	cos - return the cosine of a double value	
SYNOPSIS	<pre>#include <math.h> double cos (double -x);</pre>	
DESCRIPTION	cos() returns the sine of x . x should be specified in radians.	

TYPE	cosh (M) - Float Math	ANSI
NAME	cosh - return the hyperbolic cosine of a double value	
SYNOPSIS	<pre>#include <math.h> double cosh (double x);</pre>	
DESCRIPTION	cosh() returns the hyperbolic cosine of <i>x</i> . cosh() will return HUGE_VAL and set errno equal to ERANGE if <i>x</i> is greater than LOGHUGE.	
DIAGNOSTICS	The symbolic values are defined in math.h .	

TYPE	cotan (M) - Float Math	Aztec									
NAME	cotan - return the cotangent of a double value										
SYNOPSIS	<pre>#include <math.h> double cotan (double x);</pre>										
DESCRIPTION	cotan() returns the cotangent of <i>x</i> . <i>x</i> should be specified in radians. cotan() is not specified by ANSI and may not be portable to other compilers/architectures.										
DIAGNOSTICS	Error Codes: <table><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr><tr><td>$x < \text{TINY_VAL}$</td><td>-HUGE_VAL (if $x < 0$) HUGE_VAL (if $x > 0$)</td><td>ERANGE</td></tr><tr><td>$x > 6.74652e^9$</td><td>0.0</td><td>ERANGE</td></tr></table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	$ x < \text{TINY_VAL}$	-HUGE_VAL (if $x < 0$) HUGE_VAL (if $x > 0$)	ERANGE	$ x > 6.74652e^9$	0.0	ERANGE
<i>condition</i>	<i>return value</i>	<i>errno</i>									
$ x < \text{TINY_VAL}$	-HUGE_VAL (if $x < 0$) HUGE_VAL (if $x > 0$)	ERANGE									
$ x > 6.74652e^9$	0.0	ERANGE									

TYPE	creat (C) - UNIX I/O	Aztec
NAME	creat - create a new file	
SYNOPSIS	<pre>#include <fcntl.h> int creat(char *name, int pmode);</pre>	
DESCRIPTION	creat() creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated to 0 length (this is done by erasing and then creating the file). creat() returns as its value an integer called a "file descriptor." Whenever a call is made to one of the unbuffered I/O functions to access the file, its file descriptor must be included in the function's parameters. <i>name</i> is a pointer to a character string which is the name of the device or file to be opened. For most systems, <i>pmode</i> is optional; if specified, it is ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX creat() function requires it. In this case, <i>pmode</i> should have the octal value 0666. For some systems, <i>pmode</i> is required and has a special meaning.	
DIAGNOSTICS	If creat() fails, it returns -1 as its value and sets a code in the global integer errno.	

TYPE	ctime (C) - Time	ANSI
NAME	ctime - convert a time value to an ASCII string	
SYNOPSIS	<pre>#include <time.h> char *ctime (const time_t *timer);</pre>	
DESCRIPTION	ctime() is equivalent to the asctime() function, except that the time value should be in time_t format rather than tm format. As with asctime(), a pointer to the ASCII time string is returned.	
SEE ALSO	asctime(), time(), localtime()	

TYPE	difftime (C) - Time	ANSI
NAME	difftime - compute the difference between two times	
SYNOPSIS	<pre>#include <time.h> double difftime (time_t time1, time_t time 0);</pre>	
DESCRIPTION	difftime() returns the difference of the two time values: <i>time1</i> - <i>time0</i> . The difference is expressed in seconds and is returned as a double .	
SEE ALSO	asctime(), ctime()	

TYPE	div (C) - Integer Math	ANSI
NAME	div - compute the quotient and remainder of the division of two integers	
SYNOPSIS	<pre>#include <stdlib.h> div_t div (int numerator, int denominator)</pre>	
DESCRIPTION	div() divides two signed integers and returns both the quotient and remainder as a type <code>div_t</code>. The <code>div_t</code> type is defined in the <code>stdlib.h</code> header file as being: <pre>typedef struct { int quot; int rem; } div_t;</pre>	
SEE ALSO	ldiv()	

TYPE	dos_packet (C) - Amiga	<i>Amiga</i>
NAME	dos_packet - output packet	
SYNOPSIS	<pre>long dos_packet (port, type, arg1, arg2, arg3, arg4, arg5, arg6, arg7); struct MsgPort *port; long type, arg1, arg2, arg3, arg4, arg5, arg6, arg7;</pre>	
DESCRIPTION	dos_packet() sends a dos packet with the specified type and arguments to the specified port.	

TYPE	exec (C) - Amiga <i>Amiga</i>
NAME	execl, execv, execlp, execvp - load and start another program
SYNOPSIS	<pre> execl (char *name, char *arg0, char *arg1, ..., char *argn, 0); execv (char *name, char *argv); execlp (char *name, char *arg0, char *arg1, ..., char *argn, 0); execvp (char *name, char *argv[]); </pre>
DESCRIPTION	The exec functions can be used within the CLI environment to load and start another program. The called program is loaded on top of the calling program; thus, if the exec function succeeds, it does not return to the caller. Parameters <i>name</i> is the name of the file containing the program to be loaded. The exec functions can pass arguments to the called program. execl() and execlp() build a command line by concatenating the strings pointed at by <i>arg1</i>, <i>arg2</i>, and so on. If a C program is being called, its main function will see <i>arg0</i> as <i>argv[0]</i>, <i>arg1</i> as <i>argv[1]</i>, and so on. execv() and execvp() build a command line by concatenating the strings pointed at by <i>argv[0]</i>, <i>argv[1]</i>, and so on. The <i>argv</i> array must have a null pointer as its last entry. If a C program is being called, its main function will see the calling function's <i>argv[i]</i> as its <i>argv[i]</i>.

The Functions

execl() and **execv()** load a program from the specified file. **execl()** is useful when a fixed number of arguments is being passed to a program. **execv()** is useful for programs which are passed a variable number of arguments.

execlp() and **execvp()** will search for a program first in the current directory, then in the directories specified by the CLI Path command, and then in the C: directory.

Other Features

If an **exec()** function fails, for example because the file does not exist, **exec()** terminates the calling program.

TYPE	exit (C) - Miscellaneous	ANSI
NAME	exit, _exit - terminate calling program	
SYNOPSIS	<pre>#include <stdlib.h> void exit(int code); void _exit(int code);</pre>	
DESCRIPTION	exit() and _exit() terminate the calling program, after releasing all dynamically-allocated memory that was obtained by calling malloc(), calloc(), realloc(), or sbrk(). The functions differ in that exit() closes all files opened for standard and unbuffered I/O, while _exit() does not. If the program was started from within a CLI environment, the program's return code is set to <i>code</i>. Control then returns to a waiting program, which is usually the CLI. When the program was activated by another program's fexec() call, it is this calling program that is waiting and that resumes when the called program exits. If the program was started from the Workbench, then it was executing as an independent process. In this case, the program's process is deleted and the space associated with its stack, segment list, and process structure is released.	

TYPE	exp (M) - Float Math	ANSI									
NAME	exp - compute the exponential function e^x										
SYNOPSIS	<pre>#include <math.h> double exp (double x);</pre>										
DESCRIPTION	exp() computes the exponential function of the input parameter x and returns it.										
SEE ALSO	log(), log10(), pow(), sqrt()										
DIAGNOSTICS	Values of x which cause exp() to compute an extremely large value, i.e., greater than LOG_HUGE, will cause exp() to return the value HUGE_VAL and set the error code ERANGE in the globally defined integer errno. Error codes: <table><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr><tr><td>$x > \text{LOG_HUGE}$</td><td>HUGE_VAL</td><td>ERANGE</td></tr><tr><td>$x < \text{LOG_TINY}$</td><td>0.0</td><td>ERANGE</td></tr></table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	$x > \text{LOG_HUGE}$	HUGE_VAL	ERANGE	$x < \text{LOG_TINY}$	0.0	ERANGE
<i>condition</i>	<i>return value</i>	<i>errno</i>									
$x > \text{LOG_HUGE}$	HUGE_VAL	ERANGE									
$x < \text{LOG_TINY}$	0.0	ERANGE									

TYPE	fabs (M) - Float Math	<i>ANSI</i>
NAME	fabs - return the absolute value of a given number	
SYNOPSIS	<pre>#include <math.h> double fabs (double x);</pre>	
DESCRIPTION	fabs() returns the absolute (or positive) value of the parameter <i>x</i> . Therefore, fabs() will return 5.3 for an input of either 5.3 or -5.3.	
SEE ALSO	floor() , ceil()	

TYPE	fclose (C) - Standard I/O	ANSI
NAME	fclose - close a buffered I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fclose (FILE *stream);</pre>	
DESCRIPTION	fclose() causes the specified <i>stream</i> to be flushed and the device or file associated with the <i>stream</i> to be closed. Any data that was written to the <i>stream</i>'s output buffer but not yet written to the file or device will be written by fclose() before closing the file, and the input and output buffers used by the <i>stream</i> will be deallocated. fclose() is automatically called by exit() before exiting the program so that no devices or files are left open after the program terminates.	
SEE ALSO	fopen()	
DIAGNOSTICS	fclose() returns 0 if it is successful. If <i>stream</i> is not a valid, open stream, EOF is returned.	

TYPE	fdopen (C) - UNIX I/O	Aztec/UNIX														
NAME	fdopen - open a file or device previously opened for unbuffered I/O for standard I/O															
SYNOPSIS	#include <stdio.h> FILE *fdopen (int fd, char *mode);															
DESCRIPTION	fdopen() is used to associate a standard I/O stream with a file or device previously opened for unbuffered I/O (with open() or another unbuffered I/O function.) The mode parameter should match the mode with which the file or device was originally opened. The various modes and their meanings are: <table><tr><th>Mode</th><th>Meaning</th></tr><tr><td>a</td><td>Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.</td></tr><tr><td>ab</td><td>Same as a, except the stream is opened as binary.</td></tr><tr><td>a+</td><td>Same as a, except the stream may also be read from.</td></tr><tr><td>a+b or ab+</td><td>Same as a+, except the stream is opened as binary.</td></tr><tr><td>r</td><td>Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.</td></tr><tr><td>rb</td><td>Same as r, except the stream is opened as binary.</td></tr></table>		Mode	Meaning	a	Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.	ab	Same as a, except the stream is opened as binary.	a+	Same as a, except the stream may also be read from.	a+b or ab+	Same as a+, except the stream is opened as binary.	r	Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.	rb	Same as r, except the stream is opened as binary.
Mode	Meaning															
a	Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.															
ab	Same as a, except the stream is opened as binary.															
a+	Same as a, except the stream may also be read from.															
a+b or ab+	Same as a+, except the stream is opened as binary.															
r	Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.															
rb	Same as r, except the stream is opened as binary.															

r+	Same as r , but the stream may also be written to.
r+b or rb+	Same as r+ , except the stream is opened as binary.
w	Open a text stream for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created with a length of 0.
wb	Same as w , except the stream is opened as binary.
w+	Same as w , except the stream may also be read from.
w+b or wb+	Same as w+ , except the stream is opened as binary.

SEE ALSO

fopen(), freopen()

EXAMPLE

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main(argc, argv)
char **argv;
int argc;
{
    FILE *fp;
    int fd;
    fd = open (argv[1],
              O_WRONLY+O_CREAT+O_EXCL);
    if (fd == -1)
    {
```



```
    if (errno == EEXIST)
        printf ("file already exists\n");
    else if (errno == ENOENT)
        printf ("unable to open file \n");
    else
        printf ("open error \n");
}
else
    printf ("the file %s was opened\n",
           argv[1]);
if ((fp = fdopen(fd, "r+")) == NULL)
    printf ("can't open file for r+ \n");
else
    printf ("fdopen worked\n");
close (fd);
fclose (fp);
}
```

TYPE	feof (C) - Standard I/O	ANSI
NAME	feof - test for end-of-file in a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int feof (FILE *stream);</pre>	
DESCRIPTION	feof() is used to test for an end-of-file condition within a specified <i>stream</i>. feof() will return a non-zero value if a <i>stream</i> has reached an end of file; otherwise, it will return a zero. This function is necessary because the standard I/O functions return EOF not only for end-of-file conditions, but also if a read error occurs.	
SEE ALSO	ferror(), clearerr(), fileno(), perror()	

TYPE	<code>ferror</code> (C) - Standard I/O	ANSI
NAME	<code>ferror</code> - test for an error in a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int ferror (FILE *stream);</pre>	
DESCRIPTION	<code>ferror()</code> is used to determine if an I/O error has occurred in the specified <i>stream</i>. <code>ferror()</code> will return a non-zero value if an error has occurred in the <i>stream</i>; otherwise, it will return a 0. This function should be used in conjunction with the <code>feof()</code> function to distinguish between end-of-file and true error conditions. An error condition will persist until <code>clearerr()</code> is called or the <i>stream</i> is closed.	
SEE ALSO	<code>feof()</code> , <code>clearerr()</code> , <code>fileno()</code> , <code>perror()</code>	

TYPE	<code>fexecl, fexecv(C)</code> - Amiga Aztec
NAME	<code>fexecl, fexecv</code> -load and start another program
SYNOPSIS	<pre>fexecl (name, arg0, arg1, arg2, ..., argn, char*0) char *name, *arg0, *arg2, ..., *argn; fexecv (name, argv) char *name, *argv[]; wait()</pre>
DESCRIPTION	The <code>fexec</code> functions load and call another program. The calling program is suspended while the called program is executing; the <code>fexec</code> function returns when the called program terminates. <code>wait</code> returns as its value the return code from the <code>fexec</code>-executed program. Parameters <code>name</code> specifies the name of the file containing the program to be loaded, and optionally, the drive on which it is located and the path to it. The <code>fexec</code> functions can pass arguments to the called program. <code>fexecl()</code> builds a command line by concatenating the strings pointed at by <code>arg1</code>, <code>arg2</code>, and so on. If a C program is being called, its main function will see <code>arg1</code> as <code>argv[1]</code>, <code>arg2</code> as <code>argv[2]</code>, and so on. <code>arg0</code>, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also <code>argv[0]</code> of a called C program is always set to a pointer to a null string. Even though <code>arg0</code> isn't passed to a program when using Aztec C, it must still be specified, even if it is just a pointer in a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

forkexecv() builds a command line by concatenating the strings pointed at by *argv[1]*, *argv[2]*, and so on. The *argv* array must have a null pointer as its last entry. If a C program is being called, its main function will see the calling function's *argv[i]* as its *argv[i]*. *argv[0]*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also *argv[0]* of a called C program is always a pointer to a null string. Even though *argv[0]* isn't passed to a program when using Aztec C, it must still be specified, even if it is just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

The Functions

forkexec1() is useful when a fixed number of arguments must be passed, and **forkexecv()** is useful when a variable number of arguments must be passed.

Other Information

The **forkexec** functions load the called functions after the calling program in memory.

Files opened for unbuffered I/O in the calling program will also be open in the called program, and will have the same file descriptors.

Files opened for standard I/O in the calling program won't be open for standard I/O in the called program, although they will be open for unbuffered I/O. Thus, before a program activates another using an **forkexec** function, it should cause the buffered data for files opened for standard I/O to be written to disk, using either the **fclose()** or **fflush()** functions.

The standard input, standard output, and standard error devices are open in the called program to the same devices or files as in the calling program. For the reasons discussed above, care is needed when either the calling or the called program accesses these logical devices using standard I/O calls.

The environment of the called program is the same as that of the calling program.

See Also

The `exec` functions also load programs. Since they overlay the calling program, they allow a larger program to be loaded. They never return to the caller.

Errors

If an `exec` function fails, it returns -1 as its value after setting a code in the global integer `errno`. These codes are defined in the **Errors** Section of the **Library Overview Chapter**.

TYPE	fflush (C) - Standard I/O	ANSI
NAME	fflush - flush an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fflush (FILE *stream);</pre>	
DESCRIPTION	fflush() is used to explicitly flush an I/O stream of any data in its buffers which has not yet been written to the file or device associated with the <i>stream</i> . fflush() is automatically called by fclose() and also any write operation which outputs an end-of-line sequence or causes the <i>stream</i> 's buffer to overflow.	
SEE ALSO	fclose() , fopen()	
DIAGNOSTICS	If fflush() is successful, it returns 0. If a write error occurs, it returns EOF. If <i>stream</i> is NULL, all <i>streams</i> are flushed.	

TYPE	fgetc (C) - Standard I/O	ANSI
NAME	fgetc - return the next available character from a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fgetc (FILE *stream);</pre>	
DESCRIPTION	fgetc() returns the next character available from <i>stream</i> and advances the <i>stream</i> 's file position by 1. The character is returned as an unsigned char promoted to an int . This function is identical to getc() , except that it is implemented as a true function rather than a macro.	
SEE ALSO	fopen(), fclose(), agetc(), getc(), getchar()	
DIAGNOSTICS	If an error occurs during the read operation, or if the end-of-file is reached, fgetc() will return EOF. The functions feof() and ferror() may be used to distinguish between a true error and end-of-file.	

TYPE	fgetpos (C) - Standard I/O	ANSI
NAME	fgetpos - save the current file position for a stream	
SYNOPSIS	<pre>#include <stdio.h> int fgetpos (FILE *stream, fpos_t *pos);</pre>	
DESCRIPTION	fgetpos() saves the current file position indicator for the specified <i>stream</i> into the object pointed to by <i>pos</i>. The value stored in <i>pos</i> is suitable for use by the fgetpos() function to reposition the <i>stream</i>. fgetpos() returns 0 if successful. If it is not successful, it returns a non-zero value or sets an error code in the global integer errno.	
SEE ALSO	ftell(), fseek(), fsetpos()	

TYPE	fgets (C) - Standard I/O	ANSI
NAME	fgets - get a string of characters from a stream	
SYNOPSIS	<pre>#include <stdio.h> char *fgets (char *s, int n, FILE *stream);</pre>	
DESCRIPTION	fgets() reads in a string of characters from the specified <i>stream</i>. fgets() will place characters from <i>stream</i> into the array pointed to by <i>s</i> until <i>n-1</i> characters are read, a newline-sequence is reached, or the end-of-file is encountered. If a newline is reached, it is included in the string. fgets() will terminate the string with a null. The pointer <i>s</i> is returned by fgets() if no errors occur.	
SEE ALSO	ferror()	
DIAGNOSTICS	If end-of-file is encountered before any characters are read, the array pointed to by <i>s</i> is left unchanged, and a null pointer is returned. If a read error occurs, the array's contents should not be considered valid, and a null pointer is also returned. The ferror() and feof() functions may be used to distinguish between an error and end-of-file.	

TYPE	<code>fileno (C) - UNIX I/O</code>	<i>Aztec</i>
NAME	<code>fileno</code> - return file descriptor associated with a stream	
SYNOPSIS	<pre>#include <stdio.h> int fileno (FILE *<i>stream</i>);</pre>	
DESCRIPTION	<code>fileno()</code> returns the low-level file descriptor associated with the specified <i>stream</i> . The file descriptor can then be used with the unbuffered I/O functions (<code>open()</code> , <code>read()</code> , etc.)	
SEE ALSO	<code>feof()</code> , <code>ferror()</code> , <code>clearerr()</code> , <code>perror()</code>	

TYPE	floor (M) - Float Math	ANSI
NAME	floor - compute the largest integer not greater than a specified number	
SYNOPSIS	<pre>#include <math.h> double floor (double x);</pre>	
DESCRIPTION	floor() returns the largest integral value not greater than the input parameter <i>x</i> . This return value is expressed as a double. For example, floor() would return 5.0 if passed 5.3, and -6.0 if passed -5.3.	
SEE ALSO	fabs(), ceil()	

TYPE	fmod (M) - Float Math	ANSI
NAME	fmod - return the remainder of the double value x/y	
SYNOPSIS	<pre>#include <math.h> double fmod(double x, double y);</pre>	
DESCRIPTION	fmod() calculates the double value x modulo y . The exact remainder, called f , is calculated so that $x = iy + f$ for an integer i , and $0 < f < y$.	
SEE ALSO	ceil(), floor(), modf()	

TYPE	fopen (C) - Standard I/O	ANSI
NAME	fopen - open a specified file or device for standard I/O access	
SYNOPSIS	<pre>#include <stdio.h> FILE *fopen (const char *filename, const char *mode);</pre>	
DESCRIPTION	fopen() is used to prepare, or "open", the device or file pointed to by <i>filename</i> for access by the standard I/O functions. Once opened by fopen(), a file or device is referred to as a <i>stream</i>. If the device or file is successfully opened, fopen() returns a pointer, called a <i>file pointer</i>, to a structure of type FILE. This file pointer is then taken as a parameter by functions such as getc() or putc() to read from, write to, and generally access the stream. The first parameter to fopen() is a pointer to the name of the device or file you want to open. The other parameter passed to fopen(), <i>mode</i>, specifies how the file or device is to be accessed. The <i>mode</i> parameter defines two important variables concerning stream accessibility: read/write access and text/binary access. • Read/write accessibility determines whether the file may be read from, written to, or both. Also, the initial position within the file upon opening, and course of action if the file does not exist, may be defined. • Text/binary access determines whether the stream should be interpreted as a series of "lines" (text mode), or as raw data (binary mode). In text mode, there is no guarantee of a correspondence between the number of characters written or read and the actual position within the file, due to possible end-of-line sequence translation and other possible alterations. In binary mode, however, there is a 1:1 correspondence between characters read and the position in a file.	

As their names suggest, text mode is appropriate for handling straight text input in a portable manner, whereas binary mode deals with an unaltered data stream.

mode points to a character string terminated by a null that specifies the stream's accessibility. The various modes and their meanings are:

<i>Mode</i>	<i>Meaning</i>
a	Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.
ab	Same as a , except the stream is opened as binary.
a+	Same as a , except the stream may also be read from.
a+b or ab+	Same as a+ , except the stream is opened as binary.
r	Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
rb	Same as r , except the stream is opened as binary.
r+	Same as r , but the stream may also be written to.
r+b or rb+	Same as r+ , except the stream is opened as binary.
w	Open a text stream for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created with a length of 0.

wb	Same as w , except the stream is opened as binary.
w+	Same as w , except the stream may also be read from.
w+b or wb+	Same as w+ , except the stream is opened as binary.

TYPE	format (C) - Conversion	Aztec
NAME	format - write formatted data using a user-defined output function	
SYNOPSIS	<pre>#include <stdio.h> int format (int (*func) (), const char *fmt, va_list varg);</pre>	
DESCRIPTION	format() is used to write formatted ASCII data by calling the function <i>func</i> repeatedly with characters. <i>func</i> should take as an argument a single character, which is a character to be output. The <i>fmt</i> string should have the same format as the function printf(). In fact, printf() could be implemented as: <pre>return (format (putchar, fmt, &args);</pre> See the printf() function for a full description of format()'s conversion process.	
SEE ALSO	va_start(), printf()	

TYPE	fprintf (C, M) - Standard I/O	ANSI
NAME	fprintf - write formatted data to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fprintf (FILE *stream, const char *fmt, ...);</pre>	
DESCRIPTION	fprintf() is used to write formatted ASCII data to the specified <i>stream</i>. The <i>fmt</i> string specifies the exact output to <i>stream</i> and also determines the number of additional arguments that are required. See the printf() function for complete details on the <i>fmt</i> string.	
SEE ALSO	printf(), format(), sprintf()	

TYPE	fputc (C) - Standard I/O	ANSI
NAME	fputc - write a character to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fputc (int <i>c</i>, FILE *<i>stream</i>);</pre>	
DESCRIPTION	fputc() takes the character <i>c</i> and writes it to the specified I/O stream. Unless an I/O error occurs, fputc() returns <i>c</i> . If an error does occur, fputc() returns EOF. fputc() is identical to the putc() function.	
SEE ALSO	putchar() , putc()	
DIAGNOSTICS	fputc() returns EOF if an error occurs. The actual error code is placed in errno .	

TYPE	fputs (C) - Standard I/O	ANSI
NAME	fputs - write a string to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fputs (const char *str, FILE *stream);</pre>	
DESCRIPTION	fputs() writes the null-terminated character string pointed to by <i>str</i> to the specified <i>stream</i>. The terminating null in <i>str</i> is not written. Unlike puts(), fputs() does not write a "\n" at the end of the string. If fputs() is successful, it returns a positive value. If an error does occur, fputs() returns EOF.	
DIAGNOSTICS	EOF is returned by fputs() if an error occurs, with the error code set in errno .	

TYPE	fread (C) - Standard I/O	ANSI
NAME	fread - read from a specified standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> size_t fread(void *buffer, size_t size, size_t count, FILE *stream);</pre>	
DESCRIPTION	fread() is used to read one or more characters from <i>stream</i>. fread() will place into the buffer pointed to by <i>buffer</i> up to <i>count</i> items, with each item of size <i>size</i>. The position indicator for <i>stream</i> is advanced by the number of characters that were successfully read. fread() returns as its value the number of items (<i>count</i> if no errors occurred) successfully read from the stream, not the number of characters. See the fwrite() function for an example that uses fread().	
SEE ALSO	fwrite()	
DIAGNOSTICS	fread() returns 0 or a number less than <i>count</i> upon end-of-file or error. The functions feof() and ferror() can be used to distinguish between the two.	

TYPE	free (C) - Memory Allocation	ANSI
NAME	free - deallocate a memory block	
SYNOPSIS	<pre>#include <stdlib.h> void free (void *ptr);</pre>	
DESCRIPTION	free() deallocates a block of memory which was previously reserved with the malloc(), lmalloc(), calloc(), or realloc() functions. This allows the space pointed at by <i>ptr</i> to be used in later attempts to allocate memory. If <i>ptr</i> is a null pointer, free() does nothing. If <i>ptr</i> does not point to a block previously allocated by malloc(), calloc(), or realloc(), or has already been de-allocated by a call to free(), the results are unpredictable.	
SEE ALSO	malloc() , calloc() , realloc() , lmalloc()	

TYPE	freeseq (C) - Amiga	Amiga
NAME	freeseq - segment unload function	
SYNOPSIS	void freeseq (int (*func) ());	
DESCRIPTION	freeseq() is used with segmented (overlay) programs to unload a segment that is no longer needed but is occupying memory. The parameter <i>func</i> is the name of any function in the segment. The segment may have been automatically loaded by a function call or explicitly loaded with the segload() function.	
SEE ALSO	segload()	

TYPE	freopen (C) - Standard I/O	ANSI
NAME	freopen - reopen a stream with a new device	
SYNOPSIS	<pre>#include <stdio.h> FILE *freopen (const char *filename, const char *mode, FILE *stream);</pre>	
DESCRIPTION	freopen() is used to substitute the name or device originally associated with <i>stream</i> with the new file or device <i>filename</i>. freopen() closes the original file or device and returns <i>stream</i> as its value. freopen() is most often used to associate new devices or files to the pre-opened <i>streams</i> stdin, stdout, and stderr. In all other respects freopen() is the same as fopen().	
SEE ALSO	fopen()	
EXAMPLE	<pre>#include <stdio.h> main() { FILE *fp; fp = freopen ("dskfile", "w+", stdout); printf ("This message is going to dskfile\n"); }</pre>	

TYPE	frexp (M) - Float Math	ANSI
NAME	frexp - decompose a floating point number	
SYNOPSIS	<pre>#include <math.h> double frexp (double <i>value</i>, int *<i>exp</i>);</pre>	
DESCRIPTION	frexp() breaks a floating point number into its component mantissa and exponent. Given <i>value</i> , frexp() places the exponent-portion of <i>value</i> into the buffer pointed to by <i>exp</i> , and returns the mantissa component.	
SEE ALSO	ldexp(), modf()	

TYPE	fscanf (C,M) - Standard I/O	ANSI
NAME	fscanf - perform formatted input conversion on a specified stream	
SYNOPSIS	<pre>#include <stdio.h> int fscanf (FILE *stream, const char *fmt, ...);</pre>	
DESCRIPTION	fscanf() is equivalent to the scanf() function, with the exception that input is read from the specified <i>stream</i> rather than from <i>stdin</i> .	
SEE ALSO	scanf(), sscanf()	

TYPE	fseek (C) - Standard I/O	ANSI
NAME	fseek - reposition current location within a stream	
SYNOPSIS	<pre>#include <stdio.h> int fseek (FILE *stream, long int offset, int origin);</pre>	
DESCRIPTION	fseek() sets the current position within a file opened for standard I/O. <i>stream</i> is the stream which is to be repositioned. The exact operation of fseek() differs depending on whether the file was opened in binary or text mode. If the file was opened in binary mode, the new position, measured in characters from the beginning of the file, is obtained by adding the requested offset to the position specified by <i>origin</i>. <i>origin</i> may have the following values: • SEEK_SET - offset from the beginning of the file. • SEEK_CUR - offset from the current position in the file. • SEEK_END - offset from the end of the file. Offset may be positive or negative. If the file was opened for text mode, the use of fseek() is restricted. Either <i>offset</i> must equal 0, or <i>offset</i> must be a value previously returned by ftell(), with <i>origin</i> set to SEEK_SET. fseek() will clear the end-of-file indicator for the stream and undo the effects of ungetc() calls if successful.	
SEE ALSO	lseek(), ftell()	
DIAGNOSTICS	fseek() returns 0 if it is successful. If an error occurs, it returns a nonzero value and sets the appropriate code in errno .	

EXAMPLE

The following routine is equivalent to opening a file in a+ mode:

```
#include <stdio.h>
main()
{
    FILE *fopen(), *fp;
    if ((fp = fopen("file1",
        "r+")) == NULL)
        fp = fopen ("file1", "w+");
    fseek (fp, 0L, 2);
        /* position 1 byte past
        last character */
    fwrite ("did the seek",13,1,fp);
    fclose (fp);
}
```

TYPE	<code>fsetpos</code> (C) - Standard I/O	ANSI
NAME	<code>fsetpos</code> - set the correct file position for a stream	
SYNOPSIS	<pre>#include <stdio.h> int fsetpos (FILE *stream, const fpos_t *pos);</pre>	
DESCRIPTION	<code>fsetpos()</code> sets the file position indicator for the specified <i>stream</i> according to the value contained in the object pointed to by <i>pos</i>. <i>pos</i> should be the value obtained by a previous <code>fgetpos()</code> call. A successful call to <code>fsetpos()</code> will clear the end-of-file indicator for the <i>stream</i> as well as the effects of the <code>unsetc()</code> function on the <i>stream</i>. On success, <code>fsetpos()</code> returns zero. If an error occurs, it returns a non-zero value and sets an error code in the global integer <code>errno</code>.	
SEE ALSO	<code>ftell()</code> , <code>fseek()</code> , <code>fgetpos()</code>	

TYPE	ftell (C) - Standard I/O	ANSI
NAME	ftell - return the current file position within a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> long ftell (FILE *stream);</pre>	
DESCRIPTION	ftell() returns the current position within the specified <i>stream</i> . For binary streams, this represents the number of characters from the beginning of the file. For text streams, the number returned by ftell() may only be meaningfully used by the fseek() function. ftell() calls on a text <i>stream</i> do not necessarily reflect a measure of characters read or written to the <i>stream</i> .	
SEE ALSO	fseek(), lseek(), fgetpos()	
DIAGNOSTICS	ftell() returns -1L if an error occurs and places the error code in <i>errno</i> . If ftell() is successful, it returns the current file position.	

TYPE	ftoa (M) - Conversion	Aztec
NAME	ftoa - convert floating point number to an ASCII string	
SYNOPSIS	<pre>#include <stdlib.h> void ftoa (double <i>val</i>, char *<i>buf</i>, int <i>precision</i>, int <i>type</i>);</pre>	
DESCRIPTION	ftoa() converts a double-precision floating point number into an ASCII string. The parameter <i>val</i> is the number to be converted, and <i>buf</i> is the buffer where the string is to be placed. It is your responsibility to ensure that the area pointed to by <i>buf</i> is sufficiently large to handle the ASCII representation of <i>val</i>. The <i>precision</i> and <i>type</i> parameters control the format used to convert the number. <i>precision</i> is used to specify the number of digits to be shown to the right of the decimal point. <i>type</i> specifies the printf-like format used: 0 for "E" format, 1 for "F" format, and 2 for "G" format. See the description of the printf() function for the details of these format specifiers.	
EXAMPLE	<pre>#include <stdio.h> #include <stdlib.h> main() { double val; val = 5.75; ftoa (val, buf, 2, 0); printf ("buf = %s\n", buf); }</pre>	
SEE ALSO	atof(), atoi(), atol()	

TYPE	fwrite (C) - Standard I/O	ANSI
NAME	fwrite - write to a specified standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> size_t fwrite(const void *buf, size_t size, size_t count, FILE *stream);</pre>	
DESCRIPTION	<code>fwrite()</code> is used to write one or more characters to <i>stream</i>. <code>fwrite()</code> takes up to <i>count</i> items, each of size <i>size</i>, from the buffer pointed to by <i>buf</i>. The file position indicator for <i>stream</i> is advanced by the number of characters that were successfully written.	
SEE ALSO	<code>fread()</code>	
DIAGNOSTICS	If a write error occurs, <code>fwrite()</code> will return a number less than <i>count</i> and set an error code in <code>errno</code> .	
EXAMPLE	This is the code for reading ten integers from file 1 (see <code>fread()</code> for more information) and writing them again to file 2. It includes a simple check that there are enough two-byte items in the first file: <pre>#include <stdio.h> main() { FILE, *fp1, *fp2; char buf[50]; int size, count, i;</pre>	


```
size = 2;
count = 10;
for (i = 0; i < 50; i++)
    buf[i] = '\\0';
if ((fp1 = fopen("file1",
"r")) == NULL)
{
    printf ("You asked me
to open file1");
    printf ("but I can't\\n");
}
if ((fp2 = fopen("file2",
"w")) == NULL)
{
    printf ("You asked me to
open file2");
    printf ("but I can't\\n");
}

if (fread(buf, size, count, fp
1) != count)
    printf ("Not enough integers
in file1 \\n");
fwrite(buf, size, count, fp2);
fclose (fp1);
fclose (fp2);
}
```

TYPE	geta4 (C) - Amiga	<i>Amiga</i>
NAME	geta4 - set up a4 register	
SYNOPSIS	<code>void geta4 (void);</code>	
DESCRIPTION	<code>geta4()</code> sets up the <code>a4()</code> register that is used as the base addressing register for the small code and data models. It need only be called as the very first item in a newly created task or process. If the code for the <code>geta4()</code> function is physically 32K away from the initial task code, the call to <code>geta4()</code> will not work. In this case, you should either try to place the <code>geta4()</code> routine within 32K of the task code, or compile the task code with the large code option <code>-mc</code>.	

TYPE	getc (C) - Standard I/O	ANSI
NAME	getc - return the next available character from a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int getc (FILE *stream);</pre>	
DESCRIPTION	getc() returns the next character available from <i>stream</i> and advances the <i>stream</i> 's file position by 1. The character is returned as an unsigned char promoted to an int. The getc() function is identical to fgetc(), except getc() is defined as a macro.	
SEE ALSO	fopen(), fclose(), agetc(), fgetc(), getchar()	
DIAGNOSTICS	If an error occurs during the read operation, or if the end-of-file is reached, getc() returns EOF. The functions feof() and ferror() may be used to distinguish between a true error and end-of-file.	

TYPE	getchar (C) - Standard I/O	ANSI
NAME	getchar - return the next available character from stdin	
SYNOPSIS	<pre>#include <stdio.h> int getchar (void);</pre>	
DESCRIPTION	getchar() returns the next character from the standard input stream, stdin . It is equivalent to the call getc(stdin) .	
SEE ALSO	getc(), fgetc(), fopen(), fclose(), agetc(), putc()	
DIAGNOSTICS	If an error occurs during the read operation, or if the end-of-file is reached, getchar() returns EOF. The functions feof() and ferror() may be used to distinguish between the two cases.	

TYPE	getenv (C) - Miscellaneous	ANSI
NAME	getenv - get value of environment variable	
SYNOPSIS	<pre>#include <stdlib.h> char *getenv (const char *name);</pre>	
DESCRIPTION	getenv() returns a pointer to the character string associated with the environment variable <i>name</i>, or a null pointer if the variable is not in the environment. If the name cannot be found, a null pointer is returned. The character string is in a dynamically-allocated buffer; this buffer will be released when the next call is made to getenv(). See the Library Overview chapter for information on environment variables.	
DIAGNOSTICS	If the specified <i>name</i> cannot be found within the environment, a null pointer is returned.	

TYPE	gets (C) - Standard I/O	ANSI
NAME	gets - get a string of characters from stdin	
SYNOPSIS	<pre>#include <stdio.h> char *gets (char *buf);</pre>	
DESCRIPTION	gets() reads a string of characters from the standard input stream, stdin, into the array pointed to by <i>buf</i>. gets() reads in characters from stdin until either a newline sequence or the end-of-file is encountered. The newline sequence, if encountered, is discarded, and a null is written immediately after the last character. This differs from the operation of the fgets() function, which preserves the new line. gets() returns <i>buf</i> if no errors occur.	
SEE ALSO	ferror() , fgets() , feof()	
DIAGNOSTICS	If end-of-file is encountered before any characters are read, the array pointed to by <i>buf</i> is left unchanged, and a null pointer is returned. If an error occurs, the array's contents should be treated as invalid, and a null pointer is returned. The ferror() and feof() functions may be used to distinguish between an error and the end-of-file.	

TYPE	gmtime (C) - Time	ANSI
NAME	gmtime - convert a calendar time into coordinated universal time	
SYNOPSIS	<pre>#include <time.h> struct tm *gmtime (const time_t *timer);</pre>	
DESCRIPTION	gmtime() converts a time value in <code>time_t</code> format into <code>struct tm</code> format. The conversion is expressed relative to Greenwich mean time.	
SEE ALSO	localtime(), ctime(), asctime(), time()	

TYPE	index (C) - String Handling	<i>Aztec</i>
NAME	index - find the first occurrence of a character in a string	
SYNOPSIS	<pre>#include <string.h> char *index (char *str, char c);</pre>	
DESCRIPTION	The <code>index()</code> function returns a pointer to the first occurrence of <code>c</code> within the string pointed to by <code>str</code>. If the character is not found, then null is returned. <code>index()</code> is not supported by ANSI and generally is not a portable function. For this reason the equivalent ANSI function <code>strchr()</code> should be used whenever possible.	
SEE ALSO	<code>strchr()</code> , <code>strrchr()</code> , <code>rindex()</code>	

TYPE	<code>int_end</code> (C) - Amiga	<i>Amiga</i>
NAME	<code>int_end</code> - restore machine state from within a C interrupt handler	
SYNOPSIS	<code>void int_end(void);</code>	
DESCRIPTION	<code>int_end()</code> is used to restore the machine registers saved by the <code>int_start()</code> function from within a C interrupt handler routine. Calling <code>int_end()</code> is equivalent to the following inline assembly code: <pre>#asm movem.l (Sp)+, d2/d3/d4 #endasm</pre> <code>int_end()</code> must be the very last line within the C interrupt handling function.	
SEE ALSO	<code>int_start()</code>	

TYPE	<code>int_start (C) - Amiga</code> <i>Amiga</i>
NAME	<code>int_start</code> - set up a function as an interrupt handler
SYNOPSIS	<code>void int_start (void);</code>
DESCRIPTION	<code>int_start()</code> is used in conjunction with the function <code>int_end()</code> to enable a standard C routine to be used as an interrupt handler. Specifically, <code>int_start()</code> saves the <code>d2</code> and <code>d3</code> data registers and the <code>a4</code> address register, as well as calling the <code>geta4()</code> function. This ensures that the C routine does not destroy any register values and also that it can address its global variables properly. The <code>int_start()</code> routine is equivalent to the following in-line assembly code: <pre>#asm movem.l D2/D3/A4, -(sp) JSR geta4# #endasm</pre> <code>int_start()</code> must be the very first line from within an interrupt routine to guarantee that it will work properly.
SEE ALSO	<code>int_end()</code>

TYPE	ioctl (C) - UNIX I/O	<i>Amiga</i>
NAME	ioctl - device I/O utility	
SYNOPSIS	<pre>#include <sgtty.h> ioctl (int <i>fd</i>, int <i>cmd</i>, struct sgtyb *<i>stty</i>);</pre>	
DESCRIPTION	The Amiga version of <code>ioctl()</code> performs only one function: It can set or reset RAW mode on the selected I/O console device. <code>ioctl()</code> is only valid under Version 1.2 or greater of the Amiga operating system.	

TYPE	is... (C) - Character Type	<i>ANSI</i>
NAME	isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii, isgraph, isxdigit - character classification functions	
SYNOPSIS	<pre>#include <ctype.h> int isalpha (int c); ...</pre>	
DESCRIPTION	These functions test a character value to check if it is of a certain type. If <code>isascii()</code> is true for <code>C</code>, then non-zero (true) will be returned under the following conditions: <code>isalpha()</code> <code>c</code> is a letter <code>isupper()</code> <code>c</code> is an uppercase letter <code>islower()</code> <code>c</code> is a lowercase letter <code>isdigit()</code> <code>c</code> is a digit <code>isalnum()</code> <code>c</code> is an alphanumeric character <code>isspace()</code> <code>c</code> is a space, tab, carriage return, newline, form feed or vertical tab <code>ispunct()</code> <code>c</code> is a punctuation character <code>isprint()</code> <code>c</code> is a printing character, valued 0x20 (space) through 0x7e (tilde) <code>iscntrl()</code> <code>c</code> is a delete character (0xff) or ordinary control character (value less than 0x20) <code>isascii()</code> <code>c</code> is an ASCII character, code less than 0x100 <code>isgraph()</code> tests for any printing character except a space <code>isxdigit()</code> tests for any hexadecimal digit character 0-9, a-f, A-F	

Otherwise, a zero (false) is returned. These functions are implemented as both true functions and as macros. By default, functions are used. If you want to use the macro versions to improve performance you should either:

- `#define __C_MACROS__` before including `ctype.h`

or

- compile with the `-sm` or `-so` options

SEE ALSO

`toupper()`, `tolower()`

TYPE	labs (C) - Integer Math	<i>ANSI</i>
NAME	labs - return the absolute value of a signed long	
SYNOPSIS	<pre>#include <stdlib.h> long int labs (long int x);</pre>	
DESCRIPTION	labs() returns the absolute value of the signed long <i>x</i> .	
SEE ALSO	abs()	

TYPE	ldexp(M) - Float Math	ANSI
NAME	ldexp - multiplies a floating point number by an integral power of 2	
SYNOPSIS	<pre>#include <math.h> double ldexp(double x, int exp)</pre>	
DESCRIPTION	The ldexp function returns the value of <i>x</i> times 2 raised to the power <i>exp</i> .	

TYPE	ldiv (C) - Integer Math	ANSI
NAME	ldiv - compute the quotient and remainder of the division of two longs	
SYNOPSIS	<pre>#include <stdlib.h> ldiv_t ldiv (long int <i>numerator</i>, long int <i>denominator</i>);</pre>	
DESCRIPTION	The <code>ldiv()</code> function divides two signed long integers and retains both the quotient and remainder as a type <code>ldiv_t</code>. The <code>ldiv_t</code> type is defined in the <code>stdlib.h</code> header file as being: <pre>typedef struct { long int quot; long int rem; };</pre>	
SEE ALSO	div()	

TYPE	localtime (C) - Time	ANSI
NAME	localtime - convert a time value relative to local time	
SYNOPSIS	<pre>#include <time.h> struct tm *localtime (const time_t *timer);</pre>	
DESCRIPTION	localtime() converts the time value contained in <i>timer</i> and converts it into tm format. The conversion is done relative to the local time.	
SEE ALSO	gmtime(), ctime(), asctime(), time()	

TYPE	log (M) - Float Math	ANSI									
NAME	log - compute the natural logarithm of a number										
SYNOPSIS	<pre>#include <math.h> double log(double x);</pre>										
DESCRIPTION	log() returns as its value the natural logarithm of the input number <i>x</i> .										
SEE ALSO	exp(), log10(), pow(), sqrt()										
DIAGNOSTICS	If the input parameter <i>x</i> is negative, log returns -HUGE_VAL and sets errno to EDOM. If the input value is 0, -HUGE_VAL is returned and errno is set to ERANGE. Error codes: <table> <tr> <th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr> <tr> <td><i>x</i> == 0</td><td>-HUGE_VAL</td><td>ERANGE</td></tr> <tr> <td><i>x</i> < 0</td><td>-HUGE_VAL</td><td>EDOM</td></tr> </table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	<i>x</i> == 0	-HUGE_VAL	ERANGE	<i>x</i> < 0	-HUGE_VAL	EDOM
<i>condition</i>	<i>return value</i>	<i>errno</i>									
<i>x</i> == 0	-HUGE_VAL	ERANGE									
<i>x</i> < 0	-HUGE_VAL	EDOM									

TYPE	log10 (M) - Float Math	ANSI									
NAME	log10 - compute the logarithm of a number to base 10										
SYNOPSIS	<pre>#include <math.h> double log10 (double x);</pre>										
DESCRIPTION	log10 returns the logarithm to base 10 of the input parameter <i>x</i> .										
SEE ALSO	exp(), log(), pow(), sqrt()										
DIAGNOSTICS	If the input parameter <i>x</i> is negative, log10 will return -HUGE_VAL and set errno equal to ENOM. If the input value <i>x</i> is 0, -HUGE_VAL is returned, and errno is set to ERANGE. Error codes: <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td><i>x</i> == 0</td><td>-HUGE_VAL</td><td>ERANGE</td></tr><tr><td><i>x</i> < 0</td><td>-HUGE_VAL</td><td>EDOM</td></tr></tbody></table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	<i>x</i> == 0	-HUGE_VAL	ERANGE	<i>x</i> < 0	-HUGE_VAL	EDOM
<i>condition</i>	<i>return value</i>	<i>errno</i>									
<i>x</i> == 0	-HUGE_VAL	ERANGE									
<i>x</i> < 0	-HUGE_VAL	EDOM									

TYPE	longjmp (C) - Miscellaneous	ANSI
NAME	longjmp - execute a non-local goto	
SYNOPSIS	<pre>#include <setjmp.h> void longjmp (jmp_buf env, int retval);</pre>	
DESCRIPTION	A call to the longjmp() function restores the stack and register state saved in the last call to setjmp() with the argument <i>env</i>, and then executes a return which makes it appear that setjmp() returned the value <i>retval</i>. It is crucial that setjmp() be called with <i>env</i> before any longjmp() calls occur. If setjmp() is not called with <i>env</i> before the first longjmp(), then the results are completely unpredictable and could cause serious problems. After completion of the longjmp() call, program execution will continue as if the corresponding setjmp() call had returned <i>retval</i>. If <i>retval</i> is set to 0, then it will be forced to zero so as not to conflict with the 0 returned by the initial call to setjmp().	
SEE ALSO	setjmp()	

TYPE	<code>lseek (C) - UNIX I/O</code> <i>Aztec /UNIX</i>
NAME	<code>lseek</code> - change current position within file
SYNOPSIS	<pre>long int lseek (int <i>fd</i>, long <i>offset</i>, int <i>origin</i>);</pre>
DESCRIPTION	<code>lseek()</code> sets the current position of a file that opened for unbuffered I/O. This position determines where the next character will be read or written. <i>fd</i> is the file descriptor associated with the file. The current position is set to the location specified by the offset and origin parameters, as follows: • If <i>origin</i> is 0, the current position is set to <i>offset</i> bytes from the beginning of the file. • If <i>origin</i> is 1, the current position is set to the current position plus <i>offset</i>. • If <i>origin</i> is 2, the current position is set to the end of the file plus <i>offset</i>. The offset can be positive or negative, to position after or before the specified origin, respectively. If <code>lseek()</code> is used on a file opened as text, only an offset of 0 may be used. If <code>lseek()</code> is successful, it returns the new position in the file (in bytes from the beginning of the file).
DIAGNOSTICS	If <code>lseek()</code> fails, it returns -1 as its value and sets an error code in the global integer <code>errno</code>. <code>errno</code> is set to <code>EBADF</code> if the file descriptor is invalid. It will be set to <code>EINVAL</code> if the offset parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

```
lseek(fd, 0L, 0);
```

`lseek()` returns the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

The variable `pos` contains the current position of the end-of-file, plus one.

3. To seek backward five bytes:

```
lseek(fd, -5L, 1);
```

The third parameter, `1`, sets the origin at the current position in the file. The offset is `-5`. The new position is the origin plus the offset. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);
```

```
lseek(fd, 5L, 1);
```

```
read (fd, buf, count);
```

TYPE	malloc (C) - Memory Allocation	ANSI
NAME	malloc - allocate a block of system memory	
SYNOPSIS	<pre>#include <stdlib.h> void *malloc (size_t size);</pre>	
DESCRIPTION	malloc() allocates a block of memory <i>size</i> bytes long. The block is allocated from an area in system memory called the "heap". See the Library Overview chapter for a description of how the heap is used. The memory allocation done by malloc() is called "dynamic allocation," because the amount of memory to be reserved for storage is determined dynamically at runtime, rather than being fixed at compile time. The storage returned from malloc() should not be assumed to have been initialized to any particular value, 0 or otherwise. If the allocation is successful, malloc() returns a pointer to the requested block.	
SEE ALSO	calloc() , realloc() , free() , lmalloc()	
DIAGNOSTICS	If malloc() cannot allocate the requested <i>size</i> block, it returns a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	memchr (C) - Block Operation	ANSI
NAME	memchr - find a character within an object	
SYNOPSIS	<pre>#include <string.h> void *memchr (const void *obj, int c, size_t n);</pre>	
DESCRIPTION	memchr() searches the first <i>n</i> bytes in the object pointed to by <i>obj</i> for the character <i>c</i> . If <i>c</i> is found, memchr() returns a pointer to it. Otherwise, a NULL pointer is returned.	
SEE ALSO	strchr(), strchr()	

TYPE	memcmp (C) - Block Operation	ANSI
NAME	memcmp - compare two blocks of memory <i>n</i> bytes long	
SYNOPSIS	<pre>#include <string.h> int memcmp (const void *blk1, const void *blk2, size_t n);</pre>	
DESCRIPTION	The <code>memcmp()</code> function compares the first <i>n</i> characters in the object pointed to by <i>blk1</i> with the first <i>n</i> characters in the object pointed to by <i>blk2</i> . <code>memcmp()</code> returns a positive number, negative number, or zero, depending on whether <i>blk1</i> is greater than, less than, or equal to <i>blk2</i> .	
SEE ALSO	<code>strcmp()</code> , <code>strncmp()</code> , <code>strcoll()</code>	

TYPE	memcpy (C) - Block Operation	ANSI
NAME	memcpy - copy a block of bytes from one object to another	
SYNOPSIS	<pre>#include <string.h> void *memcpy void *dest, const void *src, size_t n);</pre>	
DESCRIPTION	memcpy() copies the first <i>n</i> bytes from the object pointed to by <i>src</i> into the object pointed to by <i>dest</i> . The two objects should <i>not</i> overlap. memcpy() returns <i>dest</i> .	
SEE ALSO	memmove(), strcpy(), strncpy()	

TYPE	memmove (C) Block Operation	ANSI
NAME	memmove - copy a block of memory	
SYNOPSIS	<pre>#include <string.h> void *memmove(void *dest, const void *source, size_t n);</pre>	
DESCRIPTION	memmove() copies a block of data from one location in memory to another location. memmove() will copy <i>n</i> characters from the object pointed to by <i>source</i> to the object pointed to by <i>dest</i>. memmove() acts as if the data pointed to by <i>source</i> is first copied into a temporary buffer before moving it into <i>dest</i>. Therefore, overlapping blocks may be used with this function. memmove() returns the value of <i>dest</i>.	
SEE ALSO	memcpy()	

TYPE	memset (C) - Block Operation	ANSI
NAME	memset - set a block of memory to a specified value	
SYNOPSIS	<pre>#include <string.h> void memset (void *obj, int c, size_t n);</pre>	
DESCRIPTION	memset() copies the value of <i>c</i> into the first <i>n</i> bytes of the object pointed to by <i>obj</i> .	
SEE ALSO	memcpy(), memcmp()	

TYPE	mktemp (C) - UNIX I/O	Amiga
NAME	mktemp - make a unique file name	
SYNOPSIS	char *mktemp (char * <i>template</i>);	
DESCRIPTION	mktemp() replaces the character string pointed at by <i>template</i> with the name of a nonexistent file and returns as its value a pointer to the string. The string pointed at by <i>template</i> should look like a filename whose last few characters are Xs with an optional imbedded period. mktemp() replaces the Xs with a letter followed by the least significant digits of the starting address of its program's data segment. The letter will be between A and Z and will be chosen such that the resulting character string is not the name of an existing file.	
DIAGNOSTICS	For a given character string, mktemp() will try to convert the string into one of 26 file names. If all of these files exist, mktemp() will replace the first character pointed at by <i>template</i> with a null character.	
SEE ALSO	tmpfile(), tmpnam()	
EXAMPLES	The following program calls mktemp() to get a character string that it can use as a filename. If the digits selected by mktemp() are 1234, then the generated name will be one of the strings abcA001.234, abcB001.234, ..abcZ001.234. If all the strings that mktemp() considers are names of existing files, mktemp() will replace the first character of the string passed to it, a in this case, with 0.	

```
#include <stdio.h>
main()
{
    char *fname, *mktemp();
    FILE *fp, fopen();
    fname=mktemp("abcXXX.XXX");
    if (!*fname){
        printf("mktemp failed");
        exit(1);
    } else
        fp=fopen(fname, "w");
    ...
}
```

TYPE	mktime (C) - Time	ANSI
NAME	mktime - convert a time value between formats	
SYNOPSIS	<pre>#includes <time.h> time_t mktime (struct tm *timeptr);</pre>	
DESCRIPTION	mktime() translates a time value represented in struct tm format into time_t format. timeptr should be a pointer to the tm format time value, and the corresponding time_t value is returned by mktime(). If timeptr is unconvertible, mktime() returns (time_t) -1.	
SEE ALSO	asctime(), ctime(), localtime(), time()	

TYPE	modf (M) - Float Math <i>ANSI</i>
NAME	modf - break a floating point value into integral and fractional parts
SYNOPSIS	<pre>#include <math.h> double modf (double <i>val</i>, double *<i>iptr</i>);</pre>
DESCRIPTION	modf() breaks the floating point number <i>val</i> into its component integral (to the left of the decimal point) and fractional (to the right of the decimal point) components. The integral portion is stored in the buffer pointed to by <i>iptr</i> , and the fractional portion is returned as the return value of modf() .
SEE ALSO	frexp() , ldexp()

TYPE	movmem(C) Block Operation	<i>Aztec</i>
NAME	movmem - copy a memory block	
SYNOPSIS	<pre>void movmem(source, dest, length) void *source, *dest; unsigned int length;</pre>	
DESCRIPTION	movmem copies length characters from the block of memory pointed to by source to the block of memory pointed to by dest . The effect of movmem is as if length bytes of source were copied to a distinct temporary area and then the temporary area copied to dest . movmem can thus be used to copy blocks of memory that overlap.	
SEE ALSO	memcpy , memmove	
APPLICATION NOTE	movmem is provided for compatibility with other Manx C compiler products. It should not be used in new programs. The memmove function should be used instead.	

TYPE	open (C) - UNIX I/O	Aztec /UNIX																
NAME	open - open device or file for unbuffered I/O																	
SYNOPSIS	#include <fcntl.h> open (char *name, int mode);																	
DESCRIPTION	open() opens a device or file for unbuffered I/O. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered I/O functions. name is a pointer to a character string which is the name of the device or file to be opened. mode specifies how your program intends to access the file. The choices are as follows: <table><tr><th>Mode</th><th>Meaning</th></tr><tr><td>O_RDONLY</td><td>read only</td></tr><tr><td>O_WRONLY</td><td>write only</td></tr><tr><td>O_RDWR</td><td>read and write</td></tr><tr><td>O_CREAT</td><td>create file, then open it</td></tr><tr><td>O_TRUNC</td><td>truncate file, then open it</td></tr><tr><td>O_EXCL</td><td>cause open() to fail if file already exists; used with O_CREAT</td></tr><tr><td>O_APPEND</td><td>position file for appending data</td></tr></table> These open modes are integer constants defined in the files fcntl.h. Although the true values of these constants can be used in a given call to open(), use of the symbolic names ensures compatibility with UNIX and other systems. The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values		Mode	Meaning	O_RDONLY	read only	O_WRONLY	write only	O_RDWR	read and write	O_CREAT	create file, then open it	O_TRUNC	truncate file, then open it	O_EXCL	cause open() to fail if file already exists; used with O_CREAT	O_APPEND	position file for appending data
Mode	Meaning																	
O_RDONLY	read only																	
O_WRONLY	write only																	
O_RDWR	read and write																	
O_CREAT	create file, then open it																	
O_TRUNC	truncate file, then open it																	
O_EXCL	cause open() to fail if file already exists; used with O_CREAT																	
O_APPEND	position file for appending data																	

are optional. They may be included by adding them to the *mode* parameter, as in the examples below.

By default, the `open` fails if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the `O_CREAT` option. If `O_EXCL` is given in addition to `O_CREAT`, the `open` will fail if the file already exists; otherwise, the file is created.

If the `O_TRUNC` option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when `O_TRUNC` is used, `O_CREAT` is not needed.

If `O_APPEND` is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to the end of the file.

If `open()` does not detect an error, it returns an integer called a "file descriptor." This value is used to identify the open file during unbuffered I/O operations. The file descriptor is very different from the file pointer which is returned by `fopen()` for use with buffered I/O functions.

DIAGNOSTICS If `open` encounters an error, it returns -1 and sets the global integer `errno` to a symbolic value which identifies the error.

EXAMPLES

1. To open the file, `testfile` for read-only access:

```
fd = open("testfile", O_RDONLY);
```

If `testfile` does not exist `open` returns -1 and sets `errno` to `ENOENT`.

2. To open the file, `testfile` for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main(argc, argv)
char **argv;
int argc;
/* should add this declaration */
{
    int fd;
    fd = open (argv[1],
O_WRONLY+O_CREAT+O_EXCL);
    if (fd == -1){
        if (errno == EEXIST)
            printf ("file already exists\n");
        else if (errno == ENOENT)
            printf ("unable to open file \n");
        else
            printf ("open error \n");
    }
    else
        printf ("the file %s was
opened\n", argv[1]);
    close (fd);
}
```

TYPE	perror (C) - Standard I/O	ANSI																						
NAME	perror - print a system error message																							
SYNOPSIS	<pre>#include <stdio.h> void perror (const char *str);</pre>																							
DESCRIPTION	When an error occurs in one of the standard math or I/O library functions, an error number indicating which error occurred is written to the global integer errno. The perror() function is used to print an error message to the stderr() stream which corresponds to the error indicated by errno. The exact format of the message written to stderr() is: • The string pointed at by <i>str</i> (only if <i>str</i> and the first character it points to are not null) followed by a colon and a space. • The system error message string followed by a new-line. On the Amiga, the possible values of errno and their corresponding error messages are: <table><tr><th><i>Errno Value</i></th><th><i>message</i></th></tr><tr><td>0</td><td>Error 0</td></tr><tr><td>ENOENT</td><td>No such file or directory</td></tr><tr><td>E2BIG</td><td>Arg list too long</td></tr><tr><td>EBADF</td><td>Bad file descriptor</td></tr><tr><td>ENOMEM</td><td>Not enough memory</td></tr><tr><td>EEXIST</td><td>File exists</td></tr><tr><td>EINVAL</td><td>Invalid argument</td></tr><tr><td>ENFILE</td><td>File table overflow</td></tr><tr><td>EMFILE</td><td>Too many open files</td></tr><tr><td>ENOTTY</td><td>Not a console</td></tr></table>		<i>Errno Value</i>	<i>message</i>	0	Error 0	ENOENT	No such file or directory	E2BIG	Arg list too long	EBADF	Bad file descriptor	ENOMEM	Not enough memory	EEXIST	File exists	EINVAL	Invalid argument	ENFILE	File table overflow	EMFILE	Too many open files	ENOTTY	Not a console
<i>Errno Value</i>	<i>message</i>																							
0	Error 0																							
ENOENT	No such file or directory																							
E2BIG	Arg list too long																							
EBADF	Bad file descriptor																							
ENOMEM	Not enough memory																							
EEXIST	File exists																							
EINVAL	Invalid argument																							
ENFILE	File table overflow																							
EMFILE	Too many open files																							
ENOTTY	Not a console																							

EACCESS	Permission denied
EIO	I/O error
ENOSPC	No space left on device
ERANGE	Result too large
EDOM	Argument out of domain
ENOEXEC	Exec format error
EROFS	Read-only file system
EXDEV	Cross-device rename
EAGAIN	Nothing to read

TYPE	pow (M) - Float Math	ANSI
NAME	pow - compute x to the y th power	
SYNOPSIS	<pre>#include <math.h> double pow (double x, double y);</pre>	
DESCRIPTION	Given two double precision numbers x and y , pow() returns the value of x to the y th power (x^y).	
SEE ALSO	exp() , log() , log10() , sqrt()	
DIAGNOSTICS	If $x = 0$ and $y \leq 0$, ENOM is set for errno and 0.0 is returned. If $x < 0$ and y is not an integral value, EDOM is set for errno and 0.0 is returned.	

TYPE	printf (C, M) - Standard I/O	ANSI
NAME	printf - formatted output function	
SYNOPSIS	<pre>#include <stdio.h> int printf (const char *fmt,...);</pre>	
DESCRIPTION	printf() is the C language's primary formatted-output function. printf() takes a series of arguments, converts them to ASCII strings, and writes the formatted information to the stdout stream. The Format String The format string <i>fmt</i> must be present in every call to printf(). This string determines exactly what gets written to stdout and may contain two types of items, ordinary characters and conversion specifiers: • Ordinary characters are always copied verbatim to the output stream. • Conversion specifiers direct printf() to take arguments from printf()'s argument list and format them. Conversion specifiers always begin with a % character. Conversion Specifiers Conversion specifiers always take the following form: <pre>% [flags] [width] [.precision] [size-mod] type</pre> where • The optional <i>flags</i> field controls output justification, sign characters on numerical values, prefixes on hex and octal numbers, decimal points, and trailing blanks.	

- The optional *width* field specifies the minimum number of characters to print (the field width), with padding done with blanks or zeros.
- The optional *precision* field specifies either the minimum number of digits to be printed for integral types, or the number of characters after the decimal point to be printed for floating-point types.
- The optional *size-mod* field specifies that the argument may be long, short, or unsigned.
- The *type* field specifies the actual type of the argument that `printf()` will be converting, such as string, integer, double, etc.

Note that all of the above fields are optional except for *type*. The following tables list the valid options for these fields. In the table, integral types are considered to be **char**, **int**, **long int**, **short int**, and unsigned versions of these types. Floating-point types are **float**, **double**, and **long double**.

Flags Field

Character *Effect on Conversion*

-	The result is left justified, with padding on the right with blanks. By default, when "-" is not specified, the result is right-justified with padding on the left with 0's or blanks.
+	The result will always have a "-" or "+" prepended to it if it is a numeric conversion.
space	Positive numbers begin with a space instead of a + character, but negative values still have a prepended "-".
#	The argument is formatted using an alternate form from the usual conversion. For x or X types, a 0x or 0X is used as a prefix to the argument. For 0 types, a 0 is always prepended to non-zero results. For e, E, and F types, the result will always

contain a decimal point, even when the number has no numbers following the decimal point. G and g types are also converted this way, with the addition that trailing zeros are not removed.

Width Field

<i>Character</i>	<i>Effect or Conversion</i>
------------------	-----------------------------

<i>n</i>	A minimum of <i>n</i> characters are output. If the conversion has less than <i>n</i> characters, the field is padded with blanks.
----------	--

<i>*</i>	The width specifier is supplied in the argument list before the actual conversion argument.
----------	---

Precision Field

<i>Character</i>	<i>Effect on Conversion</i>
------------------	-----------------------------

<i>.n</i>	A minimum of <i>n</i> characters will be provided in the field for integral conversions (d, i, o, u, x, X). For floating-point conversions (e, E, f, g, G) <i>n</i> specifies the number of digits after the decimal point. For string conversions (s), <i>n</i> is the number of characters printed from the string.
-----------	---

<i>.</i>	This is the same as <i>.n</i> where <i>n</i> is 0. No decimal point is printed.
----------	---

<i>.*</i>	The precision is supplied in the argument list before the actual conversion argument.
-----------	---

Size-mod Field

<i>Character</i>	<i>Effect on Conversion</i>
------------------	-----------------------------

<i>h</i>	The argument should be taken as a short integer. Valid only for integral conversion (d, i, o, u, x, X).
----------	---

<i>l</i>	The argument should be taken as a long int for integral conversions (d, i, o, u, x, X) and as a
----------	---

double for floating-point conversions (e, E, f, g, G).

L The argument should be taken as a long **double** for floating-point conversions (e, E, f, g, G).

Type Field

<i>Character</i>	<i>Argument Type</i>	<i>Conversion</i>
d	integral type	signed decimal integer (base 10)
i	integral type	signed decimal integer
o	integral type	unsigned octal (base 8) integer
u	integral type	unsigned decimal integer (base 16)
x	integral type	unsigned hexadecimal integer with lower case letters, i.e. a, b, c, d, e, f
X	integral type	unsigned hexadecimal integer with uppercase letters, i.e., A, B, C, D, E, F
f	floating point	signed real number with the form [-]ddd.ddd. The number of digits after the decimal point is determined by the precision field, or is 6 if precision is not specified. The "." will not appear if precision is 0.
e	floating point	Signed real number with the form [-]d.ddd e+dd. There is always exactly one digit before the decimal point, followed by an "e", followed by an exponent at least two characters long. Precision considerations are the same as "f".

E	floating point	Same as "e", but with a capital "E" for the exponent.
g	floating point	Signed real number in either "f" or "e" form. "e"-format is used if the exponent is less than -4 or greater than or equal to the precision (6 by default). Otherwise, "f" format is used.
G	floating point	Same as "g", but use "E" or "F" formats.
c	integral type	The integer argument is converted to an unsigned char, and the ASCII character corresponding to the number is output.
s	pointer	The string pointed to is written out until either a null is reached, or the number of characters written equals the precision field.
p	pointer	The contents of the pointer (that is, the address of the object is printing at) is written to the output.
n	pointer	The number of characters written so far is written to the argument, which should be a pointer to an int. No output conversion is done.
%		A % is written, and no argument is converted. The full syntax of this is "%%".

There are two versions of `printf()` in the libraries: a non-floating point version in `c.lib` and a floating point version in `m.lib`. If a `%f` or `%g` conversion prints out as "f" or "g", then you are either not linking in a math library or are linking libraries in the wrong order. To fix this problem, you should link in the math library before the c library on your link line, as shown:

```
ln ... -ln -lc
```

TYPE	putc (C) - Standard I/O	ANSI
NAME	putc - write a character to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int putc (int c, FILE *stream);</pre>	
DESCRIPTION	putc() takes the character <i>c</i> and writes it to the specified I/O stream. Unless an I/O error occurs, putc() returns <i>c</i> . If an error does occur, putc() returns EOF. This function is identical to fputc() .	
SEE ALSO	putchar(), fputc()	
DIAGNOSTICS	putc() returns EOF if an error occurs. The actual error code is placed in errno .	

TYPE	putchar (C) - Standard I/O	ANSI
NAME	putchar - write a character to the <code>stdout</code> stream	
SYNOPSIS	<pre>#include <stdio.h> int putchar (int c);</pre>	
DESCRIPTION	<code>putchar()</code> is identical to the function <code>putc()</code>, except that it always writes to <code>stdout</code>, i.e., it is the same as <pre>putc (stdout)</pre>	
DIAGNOSTICS	<code>putchar()</code> returns EOF if an I/O occurred during the write. The error code is set in the global integer <code>errno</code>.	
SEE ALSO	<code>putc()</code> , <code>fputc()</code>	

TYPE	puts (C) - Standard I/O	ANSI
NAME	puts - write a string to stdout	
SYNOPSIS	<pre>#include <stdio.h> int puts (const char *str);</pre>	
DESCRIPTION	puts writes the null-terminated character string pointed to by <i>str</i> to the stdout stream, followed by a "\n". The null at the end of the string is not written to stdout. If puts is successful, it returns a positive value. If an error occurs, EOF is returned.	
DIAGNOSTICS	EOF is returned by puts() if an error occurs, with the error number contained in errno.	

TYPE	qsort (C) - Miscellaneous	Aztec
NAME	qsort - sort an array of records in memory	
SYNOPSIS	<pre>#include <stdlib.h> void qsort (void *array, size_t num, size_t width, int (*func) ());</pre>	
DESCRIPTION	qsort() sorts an array of elements using Hoare's Quicksort algorithm. <i>array</i> is a pointer to the array to be sorted; <i>num</i> is the number of records to be sorted; <i>width</i> is the size in bytes of each array element; <i>func</i> is a pointer to a function that is called for a comparison of two array elements. This function must be written by you. <i>func</i> is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.	
EXAMPLE	The Aztec linker, ln, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.	

```
#include <stdio.h>
define MAXLINES 2000
define LINESIZE 16
char *lines[MAXLINES];
main()
{
    int i, numlines, cmp();
    char buf[LINESIZE],
    * malloc(), *gets();
    for (numlines = 0; numlines < MAXLINES;
        ++numlines)
    {
        if (gets(buf) == (char *)NULL)
            break;
        lines[numlines] = malloc(LINESIZE);
        strcpy(lines[numlines], buf);
    }
    qsort (lines, numlines,
        sizeof(char *), cmp);
    for (i = 0; i < numlines; ++i)
        printf ("%s \n", lines[i]);
}

cmp(a, b)
char **a, **b;
{
    return strcmp(*a, *b);
}
```

TYPE	ran (M) - Float Math	<i>Aztec</i>
NAME	ran - generate floating point random numbers	
SYNOPSIS	<pre>#include <math.h> double ran (void);</pre>	
DESCRIPTION	ran() returns as its value a random floating point number between 0.0 and 1.0. The seed value for ran() can be set with the function sran(). ran is not defined by ANSI.	
SEE ALSO	sran(), rand(), srand()	

TYPE	rand (C) - Integer Math	ANSI
NAME	rand - return a pseudo-random integer	
SYNOPSIS	<pre>#include <stdlib.h> int rand (void);</pre>	
DESCRIPTION	rand() returns a pseudo-random integer in the range 0 to RAND_MAX. The seed value used by rand() may be set with the srand() function.	
SEE ALSO	srand(), ran(), sran()	

TYPE	read (C) - UNIX I/O	Aztec /UNIX
NAME	read - read from a device or file using unbuffered I/O	
SYNOPSIS	<pre>#include <fcntl.h> int read (int fd, void *buf, size_t bufsize);</pre>	
DESCRIPTION	read() reads characters from a device or disk file which has been previously opened by a call to open() or creat(). In most cases, the information is read directly into the caller's buffer. <i>fd</i> is the file descriptor which was returned to the caller when the device or file was opened. <i>buf</i> is a pointer to the buffer into which the information is to be placed. <i>bufsize</i> is the number of characters to be transferred. If read() is successful, it returns as its value the number of characters transferred. If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.	
SEE ALSO	open() , close()	
DIAGNOSTICS	If the operation is not successful, read() returns -1 and places a code in the global integer errno .	

TYPE	realloc (C) - Memory Allocation	ANSI
NAME	realloc - re-allocate an existing memory block to a different size	
SYNOPSIS	<pre>#include <stdlib.h> void *realloc(void *ptr, size_t size);</pre>	
DESCRIPTION	realloc() changes the size of a memory block, <i>ptr</i>, that was originally allocated with the calloc(), realloc(), or malloc() functions. The data contained in the original block is guaranteed to be preserved by realloc(), even if the block has to be moved to accommodate a larger size. If <i>size</i> is larger than the original block size, the area beyond the original block size should not be assumed to contain any initial value. If the new size is smaller, the data will be truncated at the appropriate point. If <i>size</i> is 0, realloc() deallocates the block in a manner similar to the free() function. If <i>ptr</i> is 0, realloc() behaves like the malloc() function and allocates a new block of length <i>size</i>. If <i>ptr</i> does not point to a block previously allocated by malloc(), calloc(), or realloc(), or if <i>ptr</i> was deallocated by the free() function, the behavior of realloc() is unpredictable.	
SEE ALSO	malloc(), calloc(), free(), lmalloc()	
DIAGNOSTICS	If realloc() cannot find a block at least <i>size</i> bytes in length available, it returns a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	remove (C) - Standard I/O	ANSI
NAME	remove - delete a file	
SYNOPSIS	<pre>#include <stdio.h> int remove (const char *filename);</pre>	
DESCRIPTION	remove() deletes the disk file named <i>filename</i> from the disk. remove() returns 0 if it is successful, and non-zero if it is not.	
SEE ALSO	unlink()	
DIAGNOSTICS	If an error occurs, remove() sets errno with the error code and returns a non-zero value.	

TYPE	rename (C) - Standard I/O	ANSI
------	---------------------------	------

NAME	rename - rename a disk file
------	-----------------------------

SYNOPSIS	<pre>#include <stdio.h> int rename (const char *old, const char *new);</pre>
----------	--

DESCRIPTION	rename() changes the name of a file. <i>old</i> is a pointer to a null-terminated string containing the old file name, and <i>new</i> is a pointer to a null-terminated string containing the new name of the file.
-------------	--

If successful, **rename()** returns 0 as its value; if unsuccessful, it returns EOF.

If a file with the new name already exists, **rename()** sets E_EXIST in the global integer **errno** and returns EOF as its value without renaming the file.

TYPE	rewind (C) - Standard I/O	ANSI
NAME	rewind - reposition a stream's position indicator to the beginning of the file	
SYNOPSIS	<pre>#include <stdio.h> void rewind (FILE *stream);</pre>	
DESCRIPTION	rewind() sets the indicated <i>stream</i>'s file position indicator to the beginning of the file. rewind() is equivalent to the call: <pre>(void) fseek (stream, 0L, SEEK_SET);</pre> with the exception that rewind() also clears the error indicator for the <i>stream</i>.	
SEE ALSO	fseek()	

TYPE	<code>rindex (C) - String Handling</code> <i>Aztec</i>
NAME	<code>rindex</code> - find the last occurrence of a character in a string
SYNOPSIS	<pre>#include <string.h> char *rindex (char *str, char c);</pre>
DESCRIPTION	<code>rindex()</code> returns a pointer to the last occurrence of <i>c</i> within the string pointed to by <i>str</i>. If the character is not found in <i>str</i>, then <code>NULL</code> is returned. <code>rindex()</code> is not supported by ANSI and generally is not a portable function. For this reason, the equivalent ANSI function <code>strrchr()</code> should be used whenever possible.
SEE ALSO	<code>strchr()</code> , <code>strrchr()</code> , <code>index()</code>

TYPE	sbrk (C) - Memory Allocation	<i>Aztec</i>
NAME	sbrk - memory allocation function	
SYNOPSIS	<pre>#include <stdlib.h> void *sbrk (size_t size);</pre>	
DESCRIPTION	sbrk(), which is contained along with alternate versions of other memory-allocation functions in the file heapmem.o, provides an elementary means of allocating and deallocating space from a contiguous block of memory. When first called, sbrk() gets a block of memory by calling the Amiga function AllocMem. The default size of this block is 40K bytes. A program can specify a different size by setting the desired size in the global long _Heapsize before issuing its first call to a memory-allocation function or to a standard I/O function. sbrk() maintains a pointer to the top of allocated space within the block. When called, sbrk() increments this pointer by <i>size</i> bytes and returns the value that the pointer had on entry. When a program terminates, the block of memory that was allocated by sbrk(), if any, is automatically released.	
ERRORS	If an sbrk() request would make the sbrk() pointer go past the end of sbrk()'s block of memory, sbrk() will return -1 as its value, without modifying its pointer.	

TYPE	scanf (C,M) - Standard I/O	ANSI
NAME	scanf - perform formatted input conversion on the <code>stdin</code> stream	
SYNOPSIS	<pre>#include <stdio.h> int scanf (const char *fmt, ...);</pre>	
DESCRIPTION	<code>scanf()</code> converts a <i>stream</i> of text characters from the <i>stream</i> as directed by the control string pointed to by the <i>fmt</i> parameter and places the results in the additional pointer parameters. There must be the same number of format specifiers in the <i>fmt</i> string as pointer arguments. THE FORMAT STRING The <i>fmt</i> string controls exactly how <code>scanf()</code> will scan, convert, and store each of the input fields. The conversion string is made up of the following items: • conversion specifiers • white space (spaces, tabs, newlines) • ordinary characters The <code>scanf()</code> function works through the <i>fmt</i> string, attempting to match each control item with a portion of the input stream. During the matching process, <code>scanf()</code> fetches characters one at a time from the input. When a character is fetched which is not appropriate for the item being matched, <code>scanf()</code> pushes the character back into the stream using <code>ungetc()</code>. <code>scanf()</code> terminates when it first fails to match an item in the <i>fmt</i> string or when the end of the input stream is reached. It returns the number of matched conversion specifiers or EOF if the end of the input stream was reached.	

Matching White Space Characters

When a white space character is encountered in the control string, the `scanf()` function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the `scanf()` function proceeds to the next item in the control string.

Matching Ordinary Characters

If an ordinary character is encountered in the control string, the `scanf()` function fetches the next input character. If it matches the ordinary character, the `scanf()` function simply proceeds to the next control string item. If it does not match, the `scanf()` function terminates.

Matching Conversion Specifications

When a conversion specification is encountered in the control string, the `scanf()` function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification did not request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The `scanf()` function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the `scanf()` function simply ignores the fetched input characters and proceeds to the next control item.

Details of Input Conversion

A conversion specification consists of:

- The character %, which tells the `scanf()` function that it has encountered a conversion specification
- Optionally, the assignment suppression character *
- Optionally a field width, that is, a number specifying the maximum number of characters to be fetched for the conversion
- A conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present in a conversion specification, the `scanf()` function will fetch characters as if it were going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

- % A single % is expected in the input. No assignment is done.
- d A decimal integer is expected, the input digit string is converted to binary and the result placed in the `int` field pointed at by the current pointer argument. The corresponding argument is pointer to `int`.
- o An octal integer is expected; the corresponding pointer should point to an `int` field in which the converted result will be placed. The corresponding argument is pointer to unsigned `int`.
- x A hexadecimal integer is expected; the converted value will be placed in the `int` field pointed at by the current pointer argument. The corresponding argument is pointer to unsigned `int`.
- s A sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed to by the current pointer argument.
- c A character is expected. It is placed in the `char` field pointed at by the current pointer to character array. The

normal skip over leading white space is not done; to read a single char after skipping leading white space, use %ls. The field width parameter is ignored, so this conversion can be used only to read a single character. It matches a sequence of characters of the number specified by field width.

- [A sequence of characters, optionally preceded by white space but not terminated by white space, is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current pointer argument. The left bracket is followed by:

Optionally, a ^ or ~ character;
A set of characters;
A right bracket,].

If the first character in the set is not ^ or ~, the set specifies characters which are allowed; characters are fetched from the input until one is read which is not in the set.

If the first character in the set is ^ or ~, the set specifies characters which are not allowed; characters are fetched from the input until one is read which is in the set.

- i a signed integer, value of 0 for base argument. Corresponding argument should be pointer to int.
- u unassigned decimal integer. The argument corresponding to this item should be pointer to unsigned int.
- p reads in a hexadecimal long value which represents a pointer. The corresponding argument should be a void pointer.
- n Argument should be a pointer to an int. The number of characters read in so far from stdin is written to this pointer. Execution of %n does not increment the assignment count returned at completion of execution of the fscanf()/scanf() function.

e,f,g A floating point number is expected. The input string is converted to floating point format and stored in the float field pointed at by the current pointer argument. The input format for floating point numbers consists of an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or e followed by an optionally signed digit.

The conversion characters **d**, **o**, and **x** can be capitalized or preceded by **l** to indicate that the corresponding pointer is to a long rather than an **int**. Similarly, the conversion characters **e** and **f** can be capitalized or preceded by **l** to indicate that the corresponding pointer is to a **double** rather than a **float**.

The conversion characters **o**, **x**, and **d** can be optionally preceded by **h** to indicate that the corresponding pointer is to a **short** rather than an **int**.

SEE ALSO**fscanf(), sscanf()**

TYPE	scdir (C) - Amiga	Amiga
NAME	scdir - return the name of the next file matching pattern	
SYNOPSIS	<pre>#include <fcntl.h> char * scdir (char *pat);</pre>	
DESCRIPTION	scdir() is a function that permits you to perform wildcard expansion on filename patterns using native Amiga facilities. When scdir() is called with a pattern, it returns a pointer to a static area containing the null terminated name of the next file that matches the pattern, or zero if no more files match the pattern. Since the area containing the name is statically allocated, the name will be overwritten by subsequent calls to scdir(). Patterns have two wildcard characters. The asterisk (*) matches any number of characters. The question mark (?) matches a single character. Note that scdir() uses UNIX wild-card conventions, not AmigaDos conventions.	
EXAMPLE	<pre>main() { char *sav[100]; register char *pat; register int i; /* get all .c files on current dir */ pat = "*.c"; i = 0; while ((ptr = scdir(pat)) && i < 100) { sav[i] = malloc (strlen(ptr)+1); strcpy(sav[i++], ptr); } /* rest of program */ }</pre>	

TYPE	screen (S) - Amiga	Amiga
NAME	scr_beep, scr_bs, scr_tab, scr_lf, scr_cursup, scr_cursrt, scr_cr, scr_clear, scr_home, scr_curs, scr_eol, scr_linsert, scr_ldelete, scr_cinsert, scr_cdelete - screen manipulation functions	
SYNOPSIS	<pre>void scr_beep (void) void scr_bs (void) void scr_tab (void) void scr_lf (void) void scr_cursup (void) void scr_cursrt (void) void scr_cr (void) void scr_clear (void) void scr_home (void) void scr_eol (void) void scr_linsert (void) void scr_ldelete (void) void scr_cinsert (void) void scr_cdelete (void) void scr_curs (lin, col) void int lin, col;</pre>	
DESCRIPTION	These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, and insert and delete characters and lines. These functions can be used in conjunction with the normal standard I/O and unbuffered I/O functions to display characters on the console.	

scr_beep() rings the keyboard bell.

scr_bs() moves the cursor back one character space without modifying the character that was backspaced over.

scr_tab() moves the cursor right one tab stop.

scr_lf() moves the cursor down one line, scrolling if at the bottom of the screen.

scr_cursup() moves the cursor up without changing its column location.

scr_cursrt() moves the cursor right one character space, without modifying the character that was spaced over.

scr_cr() causes a carriage return.

scr_clear() clears the screen and homes the cursor.

scr_home() homes the cursor to the upper left hand corner of the screen.

scr_curs() moves the cursor to the line and column specified by the *lin* and *col* parameters, respectively.

scr_eol() erases the line at which the cursor is located, from the current cursor position to the end of the line.

scr_linsert() inserts a blank line at the cursor location, moving the lines below the cursor down one line.

scr_ldelete() deletes the line at the cursor location, moving the lines below the cursor up one line and placing a blank line at the bottom of the screen.

scr_cinsert() inserts a space at the cursor location, shifting right one character the characters in the line which are on the right of the cursor.

scr_cdelete() deletes the character at the cursor location, shifting left one character the characters in the line which are on the right of the cursor.

TYPE	segload (C) - Amiga	Amiga
NAME	segload - load a program segment	
SYNOPSIS	void segload (int (*func)());	
DESCRIPTION	segload() is used with segmented (overlay) programs to force an overlay segment into memory without actually calling any of the functions within it. Normally, when a function is called that is not in memory, its segment is automatically loaded and the function is then jumped to. The argument <i>func</i> should be the name of a function within the segment you want to load. The segload() function is often used by a program when it detects a sufficient amount of free memory is available to hold the entire program. segload() is called to load all of the program's segments at the start of the program, which puts all of the load delay just at the beginning.	
SEE ALSO	freeseq()	

TYPE	setbuf (C) - Standard I/O	ANSI
NAME	setbuf - associate an I/O stream with a specific buffer	
SYNOPSIS	<pre>#include <stdio.h> void setbuf (FILE *stream, char *buf);</pre>	
DESCRIPTION	setbuf() is equivalent to setvbuf() function called in the following manner: • If <i>buf</i> is not null, then setbuf() is the same as setvbuf() called with _IOFBF (fully buffered) for <i>mode</i> and BUFSIZE (defined in stdio.h) for <i>size</i>. Note that this means that your buffer <i>must</i> be at least BUFSIZE in length. • If <i>buf</i> is null, then setbuf() is the same as setvbuf() called with _IONBF (no buffering) for <i>mode</i>.	
SEE ALSO	setvbuf()	

TYPE	setenv (C) - Miscellaneous	Amiga
NAME	setenv - set environment variables	
SYNOPSIS	void setenv(char *name, char *buf);	
DESCRIPTION	This function adds or deletes environment variables to or from the pseudo-library used to implement the environment. <i>name</i> is a pointer to the environment variable to be changed and <i>buf</i> is a pointer to the new value. If <i>buf</i> points to a null string, the variable <i>name</i> will be removed from the environment. If the environment does not already exist, it will be created.	

TYPE	setjmp - Miscellaneous	ANSI
NAME	setjmp(C) - set up for a non-local goto	
SYNOPSIS	<pre>#include <setjmp.h> int setjmp (jmp_buf env);</pre>	
DESCRIPTION	setjmp() is used in conjunction with longjmp() to allow gotos between functions. This is useful for error recovery from low-level routines as well as quick returns from deeply-nested functions. The argument <i>env</i> should be declared as an instance of the type jmp_buf. setjmp() saves the current stack and register variable information in <i>env</i> so that this information may be restored upon invocation of a longjmp(), and returns a 0.	
SEE ALSO	longjmp()	

TYPE	setvbuf (C) - Standard I/O	ANSI						
NAME	setvbuf - associate an I/O stream with a specific buffer							
SYNOPSIS	<pre>#include <stdio.h> int setvbuf (FILE *stream, char *buf, int mode, size_t size);</pre>							
DESCRIPTION	setvbuf() is used when you want to explicitly assign a buffer to an I/O stream, rather than let the standard I/O system do it automatically. setvbuf() should be called after the stream has been opened, but before any I/O functions (i.e., reading and writing) have been performed on the stream. The type of buffering to be performed on the stream is determined by the <i>mode</i> argument: <table><tr><td>_IOFBF</td><td>The stream is <i>fully buffered</i>. On reads from the stream, the I/O system will attempt to fill the entire buffer if the buffer is empty before returning the requested byte(s). On writes, the buffer is written to the file only if the buffer is full.</td></tr><tr><td>_IOLBF</td><td>The stream is <i>line buffered</i>. As with _IOFBF, reads from the stream will fill the entire buffer. Writes, however, will flush the buffer if a newline is encountered as well as if the buffer is full.</td></tr><tr><td>_IONBF</td><td>The stream is <i>unbuffered</i>, and the <i>buf</i> and <i>size</i> arguments are ignored. This means that each read operation will read directly from the file, and each written operation will write directly to the file, with no intermediate buffering done.</td></tr></table> You must insure that <i>buf</i> stays in existence throughout the time the stream is open. This means that if <i>buf</i> is a local array, the stream must be closed before the function returns. Also, you		_IOFBF	The stream is <i>fully buffered</i> . On reads from the stream, the I/O system will attempt to fill the entire buffer if the buffer is empty before returning the requested byte(s). On writes, the buffer is written to the file only if the buffer is full.	_IOLBF	The stream is <i>line buffered</i> . As with _IOFBF, reads from the stream will fill the entire buffer. Writes, however, will flush the buffer if a newline is encountered as well as if the buffer is full.	_IONBF	The stream is <i>unbuffered</i> , and the <i>buf</i> and <i>size</i> arguments are ignored. This means that each read operation will read directly from the file, and each written operation will write directly to the file, with no intermediate buffering done.
_IOFBF	The stream is <i>fully buffered</i> . On reads from the stream, the I/O system will attempt to fill the entire buffer if the buffer is empty before returning the requested byte(s). On writes, the buffer is written to the file only if the buffer is full.							
_IOLBF	The stream is <i>line buffered</i> . As with _IOFBF, reads from the stream will fill the entire buffer. Writes, however, will flush the buffer if a newline is encountered as well as if the buffer is full.							
_IONBF	The stream is <i>unbuffered</i> , and the <i>buf</i> and <i>size</i> arguments are ignored. This means that each read operation will read directly from the file, and each written operation will write directly to the file, with no intermediate buffering done.							

are responsible for deallocating *buf* if it was dynamically allocated; it is not done automatically by `fclose()`. If you do deallocate *buf*, it must be deallocated after `fclose()`.

SEE ALSO

`setbuf()`

TYPE	signal (C) - Signal	ANSI
NAME	signal - define how to handle a signal	
SYNOPSIS	<pre>#include <signal.h> void (*signal (int sig, void (*func) (int))) (int);</pre>	
DESCRIPTION	signal() specifies that the signal whose number is <i>sig</i> is to be handled as defined by function. A SIGNAL is a special asynchronous event such as an operator-initiated interrupt or an arithmetic fault.	

Signals and the sig Parameter

The following list defines the symbolic values that *sig* can have and the signal associated with each value. These values are defined in **signal.h**. The signals that are currently supported are:

- SIGFPE** traps divide by 0 and overflow
- SIGILL** traps on illegal instruction
- SIGSEGV** traps illegal address or bus error

Signal Processing and the func Parameter

func defines the action to be performed on receipt of the specified signal. It can be one of three values: **SIG_DFL**, **SIG_IGN**, or a function address.

If the *func* for a signal is **SIG_DFL**, the program will be terminated by the operating system, without execution of the normal Aztec C exit code. This could result in a loss of information in files opened for standard output.

If the *func* for a signal is **SIG_IGN**, the signal will be ignored.

Any other value for *func* is assumed to be the address of a function. In this case, when the specified signal occurs, the

function will be called, passing the signal's number as the function's only argument. Before the function is called, the value of *func* for the received signal will be set to SIG_DFL.

The function associated with a signal can terminate its program, if desired, by calling `exit()` or `longjmp()`. It can also return to the program at the point of interruption by issuing a return statement.

When a program activates another program, by calling one of the `exec` or system functions, the *func* for the signals of the called program are initially set to SIG_DFL regardless of their setting in the calling program. If the called program returns to the calling program, the *func* for the signals in the calling program resume the value that they had just prior to the activation of the called program.

Return Values from Signal

If a requested change is accepted, signal returns the value that the specified signal's *func* had on entry to signal. If the change is rejected, signal returns the value SIG_ERR and the global integer `errno` is set to indicate the error. Currently, the only cause for rejection is an invalid signal number, causing `errno` to be set to EINVAL.

If the request cannot be honored, a value of SIG_ERR is returned and a positive value is stored in `errno`.

EXAMPLES

The following program calls `signal()`, so that upon receipt of a bus error the program can shut itself down in an orderly fashion, closing opened files, deleting temporary files, and so on.

```
#include <signal.h>
main()
{
    signal(SIGSEGV, shutdown)
    ... /* normal program execution */
}
shutdown(sign)
int sig;
{
    print("received signal %d\n", sig);
    ... /* termination */
    exit(1);
}
```

TYPE sin (M) - Float Math *ANSI*

NAME sin - return the sine of a **double** value

SYNOPSIS `#include <math.h>`
 `double sin (double x);`

DESCRIPTION sin() returns the sine of *x*. *x* should be specified in radians.

Error codes:

<i>condition</i>	<i>return value</i>	<i>errno</i>
$ x > 6.7465e^9$	0.0	ERANGE

TYPE **sinh (M) - Float Math** **ANSI**

NAME **sinh** - return the hyperbolic sine of a double value

SYNOPSIS `#include <math.h>`
 `double sinh (double x);`

DESCRIPTION `sinh()` returns the hyperbolic sine of *x*. `sinh()` will return `HUGE_VAL` and set `errno` equal to `ERANGE` if *x* is greater than 348.6.

Error codes:

<i>condition</i>	<i>return value</i>	<i>errno</i>
$ x > \text{LOGHUGE} + \sim 0.6932$	<code>HUGE_VAL</code>	<code>ERANGE</code>

TYPE	<code>sprintf (C, M)</code> - Conversion	ANSI
NAME	<code>sprintf</code> - write formatted data to a buffer	
SYNOPSIS	<pre>#include <stdio.h> int sprintf (char *buf, const char *fmt, ...);</pre>	
DESCRIPTION	<code>sprintf()</code> is used to write formatted ASCII data to the specified buffer <i>buf</i> . The <i>fmt</i> string specifies the exact contents of <i>buf</i> and also determines the number of additional required arguments. See the <code>printf()</code> function for complete details on the <i>fmt</i> string.	
SEE ALSO	<code>printf()</code> , <code>fprintf()</code> , <code>format()</code>	

TYPE	sqrt (M) - Float Math	ANSI
NAME	sqrt - compute the non-negative square root of a value	
SYNOPSIS	<pre>#include <math.h> double sqrt (double x);</pre>	
DESCRIPTION	sqrt() returns the nonnegative square root of the input parameter <i>x</i> .	
SEE ALSO	exp(), log(), log10(), pow()	
DIAGNOSTICS	If <i>x</i> is negative, errno is set to EDOM and sqrt() returns 0.0 as its value.	

TYPE	sran (M) - Float Math	<i>Aztec</i>
NAME	sran - set the random number seed for ran	
SYNOPSIS	<pre>#include <math.h> void sran (double <i>seed</i>);</pre>	
DESCRIPTION	sran() is used to set the seed-value for the function ran(). sran() is not defined by ANSI.	
SEE ALSO	ran(), rand(), srand()	

TYPE	srand (C) - Integer Math	ANSI
NAME	srand - initialize the seed value used by rand	
SYNOPSIS	<pre>#include <stdlib.h> void srand (unsigned int seed);</pre>	
DESCRIPTION	The value <i>seed</i> passed to srand() is used to determine the pseudo-random sequence returned by future calls to rand(). If srand() is called more than once with the same value for <i>seed</i>, then the same sequence will be repeated. If rand() is called before srand(), then rand() will behave as if srand() had been called with a <i>seed</i> of 1.	
SEE ALSO	rand(), ran(), sran()	

TYPE	sscanf (C,M) - Conversion	ANSI
NAME	sscanf - perform formatted input conversion on a buffer	
SYNOPSIS	<pre>#include <stdio.h> int sscanf (const char *str, const char *fmt, ...);</pre>	
DESCRIPTION	sscanf() is identical to the scanf() function, except that sscanf() reads from the buffer pointed to by <i>str</i> rather than from <i>stdin</i> .	
SEE ALSO	scanf(), fscanf()	

TYPE	stat (C) - UNIX I/O	<i>Amiga</i>
NAME	stat - return file information	
SYNOPSIS	<pre>#include <stat.h> #include <time.h> struct stat *stat(char *name, struct stat *buf);</pre>	
DESCRIPTION	stat() returns the attribute byte, date and time, and size of the <i>filename</i>. This information is returned in <i>buf</i>, which has the following format: <pre>struct stat { char st_attr; long st_mtime; long st_size; long st_rsize; };</pre> This structure, and the meaning of the bits in the attribute and time fields, are defined in the header files stat.h and time.h. <i>name</i> can optionally specify the full pathname where the file is located.	
ERRORS	stat() returns -1 if it fails, after setting a code in the global integer errno.	

TYPE	<code>_stkchk</code> (C) - Miscellaneous	<i>Amiga</i>
NAME	<code>_stkchk</code> - check for stack overflow	
SYNOPSIS	<code>void _stkchk (void);</code>	
DESCRIPTION	<code>_stkchk()</code> is an assembly language routine that is called at the beginning of every function that has been compiled with the stack checking option <code>-bd</code> . It attempts to determine if the current stack is below the base that is initialized by the startup code. It also checks to see if a code word placed at the bottom of the stack has been corrupted. If either event has not occurred, control is returned to the function; otherwise, the <code>_stkover()</code> function is called.	

TYPE	<code>_stkover</code> (C) - Miscellaneous	<i>Amiga</i>
NAME	<code>_stkover</code> - report stack overflow	
SYNOPSIS	<code>void _stkover (void);</code>	
DESCRIPTION	<code>_stkover()</code> is called by <code>_stkchk()</code> when a stack overflow has occurred. The routine supplied in the library resets the stack pointer to the top of the stack area and displays the following message on standard output: <pre>Stack overflow!</pre> It then calls the <code>_exit()</code> routine.	

TYPE	strcat (C) - String Handling	ANSI
NAME	strcat - concatenate two strings together	
SYNOPSIS	<pre>#include <string.h> char *strcat(char *dest, const char *src);</pre>	
DESCRIPTION	strcat() appends a copy of the string pointed to by <i>src</i> to the string pointed to by <i>dest</i>, so that the resulting length of <i>dest</i> is strlen (<i>dest</i>) + strlen (<i>src</i>). The first character in <i>src</i> will overwrite the ending null in <i>dest</i>. strcat() will then return a pointer to <i>dest</i>. It is important to note that the area pointed to by <i>dest</i> must be large enough to hold both the strings in <i>dest</i> and <i>src</i> (strlen (<i>dest</i>) + strlen (<i>src</i>) + 1) and it is your responsibility to ensure this.	
SEE ALSO	strncat() , strcpy() , strncpy()	

TYPE	strchr (C) - String Handling <i>ANSI</i>
NAME	strchr - search for the first occurrence of a character in a string
SYNOPSIS	<pre>#include <string.h> char *strchr (const char *str, int c);</pre>
DESCRIPTION	strchr() returns a pointer to the first occurrence of the character <i>c</i> in string <i>str</i>. If <i>c</i> is not contained in <i>str</i>, then strchr() returns a null <i>str</i>. The strchr() function is equivalent to the Aztec function index(), with the exception that strchr() can match a null character, whereas index() cannot.
SEE ALSO	strrchr() , index() , and rindex()

TYPE	strcmp (C) - String Handling	ANSI
NAME	strcmp - compare two strings	
SYNOPSIS	<pre>#include <string.h> int strcmp (const char *str1, const char *str2);</pre>	
DESCRIPTION	strcmp() compares the strings pointed to by <i>str1</i> and <i>str2</i>, and determines whether <i>str1</i> is greater than, less than, or equal to <i>str2</i>. Equality is determined in the following manner: • strcmp() will perform an unsigned comparison on each character in <i>str1</i> and <i>str2</i>, starting with the first character in each and advancing by one character at a time. strcmp() will stop when it finds two differing characters or it reaches the end of one of the strings. • If the end of <i>str1</i> is reached but not the end of <i>str2</i>, then <i>str1</i> is less than <i>str2</i>. • If the end of <i>str2</i> is reached before <i>str1</i>, then <i>str1</i> is greater than <i>str2</i>. • If the end of both strings is reached simultaneously with no differing characters, then the two strings are equal. • If the two characters differ, then <i>str1</i> is greater than <i>str2</i> if <i>str1</i>'s character is greater than <i>str2</i>. Otherwise, <i>str1</i> is less than <i>str2</i>. It should be remembered that strcmp() does numerical comparisons, not character comparisons, so that the character "a" (ASCII 97) is considered greater than the character "Z" (ASCII 90). The value returned by strcmp() is:	

- less than 0 (negative) if *str1* is less than *str2*.
- equal to 0 if *str1* is equal to *str2*.
- greater than 0 (positive) if *str1* is greater than *str2*.

SEE ALSO**strncmp()**

TYPE	strcoll (C) - String Handling	ANSI
NAME	strcoll - compare two strings using the current locale	
SYNOPSIS	<pre>#include <string.h> int strcoll (const char *str1, const char *str2</pre>	
DESCRIPTION	The strcoll() function is equivalent to the strcmp() function, with the exception that strcoll() will use the current locale for character translation. (strcoll() is truly equivalent to strcmp() , as Aztec C currently supports only the C locale.)	
SEE ALSO	strcmp()	

TYPE	strcpy (C) - String Handling	ANSI
NAME	strcpy - copy one string to another	
SYNOPSIS	<pre>#include <string.h> char *strcpy (char *dest, const char *src);</pre>	
DESCRIPTION	strcpy() copies the characters in the string pointed to by <i>src</i> to the area pointed to by <i>dest</i> , including the terminating null character. The area pointed to by <i>dest</i> must be at least (strlen(<i>src</i>) + 1) characters long. strcpy() always returns <i>dest</i> .	
SEE ALSO	strncpy(), memcpy(), memmove(), strlen()	

TYPE	strcspn (C) - String Handling	ANSI
NAME	strcspn - return the index of the first character in a string <i>str1</i> which is contained in a string <i>str2</i>	
SYNOPSIS	<pre>#include <string.h> size_t strcspn (const char *str1 const char *str2);</pre>	
DESCRIPTION	The strcspn() function returns the index of the first character in the string pointed to by <i>str1</i> which matches a character in the string pointed to by <i>str2</i> . This index is equal to the number of initial characters in <i>str1</i> which do not match a character in <i>str2</i> .	
SEE ALSO	strspn(), strpbrk(), strstr(), strtok()	

TYPE	strlen (C) - String Handling	ANSI
NAME	strlen - return the length of a string	
SYNOPSIS	<pre>#include <string.h> size_t strlen (const char *str);</pre>	
DESCRIPTION	strlen() returns the number of characters in the string pointed to by <i>str</i>, not including the terminating null character. You should be careful in allocating space for strings based on strlen(). A common mistake is to code the following: <pre>ptr = malloc (strlen ("STRING"); strcpy (ptr, "STRING");</pre> This will only set aside six characters for "STRING", when seven are actually needed (the six characters in the word plus the ending null), and the strcpy() that follows the malloc() call will write too many characters. The correct way to use strlen() is: <pre>ptr = malloc (strlen ("STRING") +1); strcpy (ptr, "STRING");</pre>	
SEE ALSO	strcpy(), strncpy()	

TYPE	strncat (C) - String Handling	ANSI
NAME	strncat - concatenate two strings together, up to <i>max</i> characters	
SYNOPSIS	<pre>#include <string.h> char *strncat(<i>dest</i>, <i>src</i> <i>max</i>);</pre>	
DESCRIPTION	strncat() appends a copy of the string pointed to by <i>src</i> to the string pointed to by <i>dest</i> until either <i>max</i> characters have been appended or a null is reached in <i>src</i>, whichever comes first. The maximum resulting length of the string pointed to by <i>dest</i> will be strlen(<i>dest</i>) and <i>max</i>. The first character in <i>src</i> overwrites the ending null in <i>dest</i>. strncat() always returns a pointer to <i>dest</i>. It is your responsibility to ensure that the area pointed to by <i>dest</i> is at least (strlen(<i>dest</i>) + <i>max</i> + 1) characters long.	
SEE ALSO	strcat() , strcpy() , strncpy()	

TYPE	strncmp (C) - String Handling	ANSI
NAME	strncmp - compare two strings, up to <i>max</i> characters.	
SYNOPSIS	<pre>#include <string.h> int strncmp (const char *str1, const char *str2, size_t max);</pre>	
DESCRIPTION	strncmp() is equivalent to the strcmp() function, with the exception that strncmp() will only compare a maximum of <i>max</i> characters. strncmp() returns a positive number, negative number, or zero, depending on whether <i>str1</i> is greater than, less than, or equal to <i>str2</i>. See the strcmp() function for more details.	
SEE ALSO	strcmp(), memcmp(), strcoll(), strxfrm()	

TYPE	strncpy (C) - String Handling	ANSI
NAME	strncpy - copy up to <i>max</i> characters from one string to another	
SYNOPSIS	<pre>#include <string.h> char *strncpy (char *dest, const char *src, size_t max);</pre>	
DESCRIPTION	strncpy() copies a maximum of <i>max</i> characters from the string pointed to by <i>src</i> to the area pointed to by <i>dest</i> . If <i>src</i> is less than <i>max</i> characters long, then <i>dest</i> is padded with null characters until <i>max</i> total characters have been written. The area pointed to by <i>dest</i> must be at least <i>max</i> characters in length.	
SEE ALSO	strcpy(), memcpy(), memmove(), strlen()	

TYPE	strpbrk (C) - String Handling	ANSI
NAME	strpbrk - return the first occurrence in a string <i>str1</i> of a character in a string <i>str2</i> .	
SYNOPSIS	<pre>#include <string.h> char *strpbrk (const char *str1, const char *str2);</pre>	
DESCRIPTION	The strpbrk() function returns a pointer to the first character in the string pointed to by <i>str1</i> which is contained in the string pointed to by <i>str2</i> . Null is returned if no characters within <i>str1</i> match those in <i>str2</i> .	
SEE ALSO	strspn(), strcspn(), strstr(), strtok()	

TYPE	strchr (C) - String Handling	ANSI
NAME	strchr - search for the last occurrence of a character in a string	
SYNOPSIS	<pre>#include <string.h> char *strchr (const char *str, int c);</pre>	
DESCRIPTION	strchr() returns the last occurrence of the character <i>c</i> in string <i>str</i>. If <i>c</i> is not contained in <i>str</i>, then strchr() returns a null. The strchr() function is equivalent to the Aztec function rindex(), with the exception that strchr() can match a null character, whereas rindex() cannot.	
SEE ALSO	strchr() , index() , and rindex()	

TYPE	strspn (C) - String Handling	ANSI
NAME	strspn - return the index of the first character in a string <i>str1</i> which is not contained in a string <i>str2</i>	
SYNOPSIS	<pre>#include <string.h> size strspn (const char *str1, const char *str2);</pre>	
DESCRIPTION	The <code>strspn()</code> function returns the index of the first character in <i>str1</i> which is not contained in the string pointed to by <i>str2</i> . This index is equal to the number of initial characters in <i>str1</i> which do match characters in <i>str2</i> .	
SEE ALSO	strcspn(), strpbrk(), strstr(), strtok()	

TYPE	strstr (C) - String Handling	ANSI
NAME	strstr - return a pointer of the first occurrence of a string <i>str2</i> within a string <i>str1</i>	
SYNOPSIS	<pre>#include <string.h> char *strstr(const char *str1, const char *str2);</pre>	
DESCRIPTION	strstr() returns the first occurrence of the substring <i>str2</i> contained within the string <i>str1</i> . If <i>str2</i> is not contained within <i>str1</i> , then null is returned.	
SEE ALSO	stcspn(), strspn(), strpbrk(), strtok()	

TYPE	strtod (C) - Conversion	ANSI
NAME	strtod - convert a string to a double	
SYNOPSIS	<pre>#include <stdlib.h> double strtod (const char *nptr, char **endptr);</pre>	
DESCRIPTION	strtod() converts the initial portion of the string pointed to by <i>nptr</i> into a double value, which is returned by strtod(). The string must be in the form: [ws] [+/-] ddd [.][ddd] [exp] where • <i>ws</i> is whitespace (newline, space, tab) • +/- means + or - • <i>ddd</i> are digits 0-9 • <i>exp</i> is an optional exponent of the form: [e/E +/- ddd] Note: In the above input, the slashes indicate "or". strtod stops reading characters when it encounters a character in <i>nptr</i> which cannot be interpreted as part of a double value. It sets <i>*endptr</i> equal to a pointer to this character (as long as <i>endptr</i> is not null).	
DIAGNOSTICS	strtod will return HUGE_VAL if the string causes overflow.	
SEE ALSO	atof(), strtoul(), strtol(), atol(), atoi()	

TYPE	strtok (C) - String Handling	ANSI
NAME	strtok - tokenize a string	
SYNOPSIS	<pre>#include <string.h> char *strtok (char *str, const char *del_list);</pre>	
DESCRIPTION	strtok() is designed to be called multiple times to break <i>str</i> into a series of tokens, each of which is delimited by a character contained in <i>del_list</i>. The operation of strtok() is detailed below. First Invocation: On the first invocation of strtok() with <i>str</i>, strtok() will search for the first character in <i>str</i> which is not contained in <i>del_list</i> (that is, it scans past delimiters.) • If a character is found which is not in <i>del_list</i>, it is considered to be the start of the first token. • If such a character is not found, there are no tokens in <i>str</i> and a null pointer is void. strtok() then searches from the token's start for a character that is contained in <i>del_list</i> (it scans to the next delimiter.) • If a character is found in <i>str</i> which is contained in <i>del_list</i>, strtok() overwrites it with a null character which terminates the token. strtok() will then internally save a pointer to the following character, from which the next token search will start. A pointer to the token is then returned by strtok(). • If no characters in <i>str1</i> are found to be in <i>del_tist</i>, then the current token is considered to extend to the end of <i>str</i>. strtok() will return a pointer to the token, and subsequent calls to strtok() will return a null pointer.	

Subsequent Invocation:

All calls of **strtok()** following the initial call should pass a null pointer for the first argument. This instructs **strtok()** to use its internal pointer in order to locate the next token. **strtok()** will then behave as described above. Subsequent calls may, if you wish, have different delimiters specified by *del_list* .

SEE ALSO

strchr(), strrchr(), strspn(), strcspn()

TYPE	strxfrm (C) - String Handling	ANSI
NAME	strxfrm - transform a string to match the current locale	
SYNOPSIS	<pre>#include <string.h> size_t strxfrm (char *dest, const char *src, size_t n);</pre>	
DESCRIPTION	strxfrm() transforms the string pointed at by <i>src</i> to match the current locale. The transformed string is then copied to <i>dest</i>, and the length of <i>dest</i> is returned. Because only the "c" locale is currently supported, strxfrm() actually does no transformations, but simply performs a strcpy() followed by a strlen().	
SEE ALSO	strcpy(), strcoll(), strcmp(), strlen()	

TYPE	tan (M) - Float Math	ANSI						
NAME	tan - return the tangent of a double value							
SYNOPSIS	<pre>#include <math.h> double tan(double x);</pre>							
DESCRIPTION	tan() returns the tangent of <i>x</i>. <i>x</i> should be specified in radians. Error codes: <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td>$x \geq 6.74652e^9$</td><td>0.0</td><td>ERANGE</td></tr></tbody></table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	$ x \geq 6.74652e^9$	0.0	ERANGE
<i>condition</i>	<i>return value</i>	<i>errno</i>						
$ x \geq 6.74652e^9$	0.0	ERANGE						

TYPE	tanh (M) - Float Math	<i>ANSI</i>
NAME	tanh - return the hyperbolic tangent of a double value	
SYNOPSIS	<pre>#include <math.h> double tanh (double x);</pre>	
DESCRIPTION	tanh() returns the hyperbolic tangent of <i>x</i> .	

TYPE	time (C) - Time	ANSI
NAME	time - return the time of day	
SYNOPSIS	<pre>#include <time.h> time_t time(time_t *timeptr);</pre>	
DESCRIPTION	The <code>time()</code> function places the current time of day into the location pointed to by <i>timeptr</i> . This value is suitable for use by the <code>ctime()</code> , <code>gmtime()</code> , and <code>localtime()</code> functions. The current time is also returned by <code>time()</code> , in the same format as is placed in <i>timeptr</i> .	
SEE ALSO	<code>ctime()</code> , <code>gmtime()</code> , <code>localtime()</code>	

TYPE	tmpfile (C) - Standard I/O	ANSI
NAME	tmpfile - create a temporary file	
SYNOPSIS	<pre>#include <stdio.h> FILE *tmpfile(void);</pre>	
DESCRIPTION	tmpfile() creates a temporary binary file and opens it for standard I/O in update (wb+) mode. tmpfile() returns as its value the file's FILE pointer. When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file is automatically deleted. If the file cannot be created, the function returns a null pointer.	
SEE ALSO	tmpnam(), mktemp()	

TYPE	tmpnam (C) - Standard I/O	<i>ANSI</i>
NAME	tmpnam - create a name for a temporary file	
SYNOPSIS	<pre>#include <stdio.h> char *tmpnam (char *s);</pre>	
DESCRIPTION	tmpnam() creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file. <i>s</i> optionally points to an area into which the name will be generated. This must contain at least L_tmpnam bytes, where L_tmpnam is a constant defined in stdio.h. <i>s</i> can also be a null pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time tmpnam() is called with a null argument. tmpnam() can be called a maximum of TMP_MAX times.	
SEE ALSO	tmpfile(), mktemp()	

TYPE	tolower (C) - Conversion	<i>ANSI</i>
NAME	tolower - convert a character to lowercase	
SYNOPSIS	<pre>#include <ctype.h> int tolower (int c);</pre>	
DESCRIPTION	tolower() converts an uppercase character to lowercase. If the value of <i>c</i> is an uppercase character, tolower() returns its lowercase equivalent, otherwise <i>c</i> is returned unchanged.	
SEE ALSO	toupper()	

TYPE	toupper (C) - Conversion	<i>ANSI</i>
NAME	toupper - convert a character to uppercase	
SYNOPSIS	<pre>#include <ctype.h> int toupper(int c);</pre>	
DESCRIPTION	toupper() converts a lowercase character to uppercase. If the value of the argument <i>c</i> is a lowercase character, toupper() returns its uppercase equivalent as its value, otherwise <i>c</i> is returned unchanged.	
SEE ALSO	tolower()	

TYPE	<code>ungetc</code> (C) - Standard I/O	ANSI
NAME	<code>ungetc</code> - push a character back into input stream	
SYNOPSIS	<pre>#include <stdio.h> int ungetc(int c, FILE *stream);</pre>	
DESCRIPTION	<code>ungetc()</code> pushes the character value of the argument <i>c</i> back onto an input <i>stream</i>. That character will be returned by the next <code>getc()</code> call on that <i>stream</i>. <code>ungetc()</code> returns <i>c</i> as its value. Only one character of pushback is guaranteed. EOF cannot be pushed back.	
DIAGNOSTICS	<code>ungetc()</code> returns EOF (-1) if the character cannot be pushed back.	

TYPE	unlink (C) - UNIX I/O	<i>Aztec/UNIX</i>
NAME	unlink - erase file	
SYNOPSIS	<pre>int unlink (<i>name</i>); char *<i>name</i>;</pre>	
DESCRIPTION	unlink() erases a file, where <i>name</i> is a pointer to a character array containing the name of the file to be erased. unlink() returns a 0 value if successful. Since unlink() is not defined by ANSI, it is strongly recommended that the equivalent ANSI function remove() be used in its place.	
DIAGNOSTICS	unlink() returns -1 if it could not erase the file and places a code in the global integer <i>errno</i> describing the error.	

TYPE	va_arg, va_end, va_start (C) - Variable Arguments <i>ANSI</i>
NAME	va_arg, va_end, va_start - variable argument access
SYNOPSIS	<pre>#include <stdarg.h> type va_arg (va_list argptr, type); void va_end (va_list argptr); void va_start (va_list argptr, parm N);</pre>
DESCRIPTION	These three macros are used as a mechanism to access individual arguments from within a function that accepts a variable number of arguments. A function which receives a variable argument count should declare a variable, <i>argptr</i>, of <i>type va_list</i>. This variable will be used as a pointer to the current argument being processed. The va_start() macro should be called first to initialize <i>argptr</i> to point to the first argument in the list. the <i>parm N</i> parameter should be the last fixed argument passed to the function. Once va_start() has been called, va_arg() may be called repeatedly to fetch arguments sequentially from the argument list. The <i>type</i> parameter to va_arg() should be the C-data <i>type</i> of the next argument (i.e., int, long, char x, etc.). va_arg() returns the value of the argument. When variable-argument processing is completed, the va_end() macro should be called.

TYPE	<code>fprintf</code> (C) - Standard I/O	ANSI
NAME	<code>fprintf</code> - write formatted ASCII data to a stream	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int fprintf (FILE *stream, const char *fmt, va_list args);</pre>	
DESCRIPTION	<code>fprintf()</code> is equivalent to the <code>fprintf()</code> function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the <code>va_start()</code> macro before the call to <code>fprintf()</code> is made. The return value of <code>fprintf()</code> is equal to the number of characters written or is a negative value if a write error occurs.	
SEE ALSO	<code>vfprintf()</code> , <code>vsprintf()</code> , <code>printf()</code> , <code>va_start()</code>	

TYPE	vprintf (C) - Standard I/O	ANSI
NAME	vprintf - write formatted ASCII data to stdout	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int vprintf (FILE *stream, const char *fmt, va_list args)</pre>	
DESCRIPTION	vprintf() is equivalent to the printf() function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the va_start() macro before the call to vprintf() is made. The return value of vprintf() is equal to the number of characters written or is a negative value if an error occurred in output.	
SEE ALSO	fprintf(), vsprintf(), printf(), va_start()	

TYPE	vsprintf (C) - Conversion	<i>ANSI</i>
NAME	vsprintf - write formatted ASCII data to a buffer	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int vsprintf(char *buf, const char *fmt, va_list args);</pre>	
DESCRIPTION	vsprintf() is equivalent to the sprintf() function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the va_start() macro before the call to vsprintf() is made. The return value of vsprintf() is the number of characters written, not including the terminating null characters.	
SEE ALSO	fprintf(), vprintf(), printf(), va_start()	

TYPE	<code>_wb_parse</code> (C) - Amiga	<i>Amiga</i>
NAME	<code>_wb_parse</code> - open standard I/O window	
SYNOPSIS	<pre>void _wb_parse (struct Process *pp, struct WBStartup *wbm);</pre>	
DESCRIPTION	<code>_wb_parse()</code> is called from the <code>_main()</code> routine and is used to open a window for standard I/O to use. The window is actually defined by setting the ToolType, WINDOW, to the desired window specification. If this is not required, this routine may be replaced by a stub in your main program. Note that even if this code is called by <code>_main()</code> , if the WINDOW tool type is not defined, there will be no window.	
EXAMPLE	<pre>WINDOW=CON:0/0/640/200/Test Window</pre>	

TYPE	write (C) - UNIX I/O	<i>Aztec /UNIX</i>
NAME	write - write to a file or device using unbuffered I/O	
SYNOPSIS	<pre>size_t write(int fd, int bufsize, char *buf);</pre>	
DESCRIPTION	write() writes characters to a device or disk which has been previously opened by a call to open() or creat(). The characters are written to the device or file directly from the caller's buffer. <i>fd</i> is the file descriptor which was returned to the caller when the device or file was opened. <i>buf</i> is a pointer to the buffer containing the characters to be written. <i>bufsize</i> is the number of characters to be written. If the operation is successful, write() returns as its value the number of characters written.	
SEE ALSO	open(), close(), read()	
DIAGNOSTICS	If the operation is unsuccessful, write() returns -1 and places a code in the global integer errno .	

Chapter 4 - Workbench

Amiga Reference Guides

This chapter only lists the Amiga Workbench function's parameters, types, and the type of value returned. For detailed information, refer to:

- *Amiga ROM Kernel Manual, Volumes 1 and 2.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Libraries and Devices.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Exec.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Hardware Reference Manual.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Intuition Reference Manual.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)

These manuals should be available through your local bookstore. Refer to the section "Suggestions for Further Reading" at the front of your *Aztec C User Guide* for a listing of other helpful resources.

If you are using Amiga Workbench routines, you should read material relating to the Amiga Workbench functions, the **functions.h** header, and direct resident library calls, in the **Getting Started** Chapter of the *Aztec C User Guide* and the **Compiler** Chapter of the *Aztec C Reference Manual*.

Amiga Workbench Functions

```
void AbortIO(ioRequest)
    struct IORequest *ioRequest;
long ActivateGadget(gadgets, window, req)
    struct Gadgets *gadgets;
    struct Window *window;
    struct Requester *req;
void ActivateWindow(window)
    struct Window *window;
void AddAnimOb(anOb, anKey, rPort)
    struct AnimOb *anOb, **anKey;
    struct RastPort *rPort;
void AddBob(bob, rPort)
    struct Bob *bob;
    struct RastPort *rPort;
void AddConfigDev(configdev)
    struct ConfigDev *configdev;
void AddDevice(device)
    struct Device *device;
long AddDosNode(bootpri, flags,*dosnode)
    long bootpri, flags;
    struct DeviceNode *dosnode;
void AddFont(textFont)
    struct TextFont *textFont;
long AddFreeList(free, mem, len)
    struct FreeList *free;
    char *mem;
    long len;
```

```
long AddGadget(pntr, gadget, position)
    struct Window *pntr;
    struct Gadget *gadget;
    long position;
long AddGList(addptr, gadget, pos, numgad, req)
    struct Window *addptr;
    struct Gadget *gadget;
    long pos, numgad;
    struct Requester *req;
void AddHead(list, node)
    struct List *list;
    struct Node *node;
void AddIntServer(intNum, interrupt)
    long intNum;
    struct Interrupt *interrupt;
void AddLibrary(library)
    struct Library *library;
long AddMemList(size, attrib, pri, base, name)
    long size, attrib, pri;
    void *base;
    char *name;
void AddPort(port)
    struct MsgPort *port;
void AddResource(resource)
    struct MiscResource *resource;
void AddSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void AddTail(list, node)
    struct List *list;
    struct Node *node;
void AddTask(task, initialPC, finalPC)
    struct Task *task;
    void *initialPC, *finalPC;
```

```
void AddTime(dest, source)
    struct timeval *dest, *source;
void AddVSprite(vs, rPort)
    struct VSprite *vs;
    struct RastPort *rPort;
long Alert(alertNum, parameters)
    long alertNum, parameters;
void *AllocAbs(bytSiz, location)
    long bytSiz;
    void *location;
long AllocBoardMem(slotspec)
    long slotspec;
long AllocCList(cLPool)
    long cLPool;
struct ConfigDev *AllocConfigDev()
struct MemList *AllocEntry(memList)
    struct MemList *memList;
long AllocExpansionMem(numslots, SAlign, SOffset)
    long numslots, SAlign, SOffset;
void *AllocMem(byteSize, requirements)
    long byteSize, requirements;
long AllocPotBits(bits)
    long bits;
void AllocRaster(width, height)
    long width, height;
void *AllocRemember(rememberKey, size, flags)
    struct Remember **rememberKey;
    long size, flags;
long AllocSignal(signalNum)
    long signalNum;
long AllocTrap(trapNum)
    long trapNum;
struct WBOject *AllocWBOject()
```

```
void *Allocate(freeList, byteSize)
    struct MemHeader *freeList;
    long byteSize;
void AlohaWorkbench(wbPort)
    struct MsgPort *wbPort;
void AndRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long AndRegionRegion(src, dst)
    struct Region *src, *dst;
void Animate(key, rPort)
    struct AnimOb **key;
    struct RastPort *rPort;
long AreaCircle(rp, cx, cy, r)
    struct RastPort *rp;
    long cx, cy, r;
short AreaDraw(rp, x, y)
    struct RastPort *rp;
    long x, y;
long AreaEllipse(rp, cx, cy, a, b)
    struct RastPort *rp;
    long cx, cy, a, b;
void AreaEnd(rp)
    struct RastPort *rp;
long AreaMove(rp, x, y)
    struct RastPort *rp;
    long x, y;
void AskFont(rp, textAttr)
    struct RastPort *rp;
    struct TextAttr *textAttr;
long AskSoftStyle(rp)
    struct RastPort *rp;
```

```
long AttemptLockLayerRom(layer)
    struct Layer *layer;
long AttemptSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
long AutoRequest(window, body, positive, negative, posFlags,
    negFlags, width, height)
    struct Window *window;
    struct IntuiText *body, *positive, *negative;
    long posFlags, negFlags, width, height;
long AvailFonts(buffer, bufBytes, types)
    char *buffer;
    long bufBytes, types;
long AvailMem(requirements)
    long requirements;
void BeginIO(ioRequest)
    struct IORequest *ioRequest;
void BeginRefresh(window)
    struct Window *window;
long BeginUpdate(layer)
    struct Layer *layer;
long BehindLayer(li, layer)
    struct Layer_Info *li;
    struct Layer *layer;
long BlitBitMap(srcBM, srcX, srcY, dstBM, dstX, dstY, sizX, sizY,
    minTerm, mask, tempA)
    struct BitMap *srcBM, *dstBM;
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm, mask;
    char *tempA;
long BlitBitMapRastPort(srcBitMap, srcX, srcY, dstRPort,
    dstX, dstY, sizX, sizY, minTerm)
    struct BitMap *srcBitMap;
    struct RastPort *dstRPort;
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm;
```

```
void BltClear(memBlock, byteCount, flags)
    char *memBlock;
    long byteCount, flags;
long BltMaskBitMapRastPort(srcBitMap, srcX, srcY, dstRPort,
    dstX, dstY, sizX, sizY, minTerm, bltmask)
    struct BitMap *srcBitMap ;
    struct RastPort *dstRPort
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm;
    void *bltmask;
void BltPattern(rp, buf, x1, y1, maxX, maxY, byteCnt)
    struct RastPort *rp;
    char *buf;
    long x1, y1, maxX, maxY, byteCnt;
void BltTemplate(src, srcX, srcMod, dstRastPort, dstX, dstY,
    sizX, sizY)
    char *src;
    struct RastPort *dstRastPort;
    long srcX, srcMod, dstX, dstY, sizX, sizY;
void BNDYOFF(rp)
    struct RastPort *rp;
struct Window *BuildSysRequest(wind, body, positive,
    negative, flags, width, height)
    struct Window *wind;
    struct IntuiText *body, *positive, *negative;
    long flags, width, height;
char *BumpRevision(newbuf, oldname)
    char *newbuf, *oldname;
void CBump(copperlist)
    struct UCopList *copperlist;
struct InputEvent *CDInputHandler(events, consoleDevice)
    struct InputEvent *events;
    struct Device *consoleDevice;
```

```
void CEND(c)
    struct UCopList *c;
long Chk_Abort()
void CINIT(c, n)
    struct UCoplist *c;
    long n;
void CMove(c, a, v)
    struct UCopList *c;
    long *a, v;
void CWait(c, v, h)
    struct UCopList *c;
    long v, h;
void Cause(interrupt)
    struct Interrupt *interrupt;
void ChangeSprite(vp, s, newdata)
    struct ViewPort *vp;
    struct SimpleSprite *s;
    short *newdata;
struct IORequest *CheckIO(ioRequest)
    struct IORequest *ioRequest;
long ClearDMRequest(window)
    struct Window *window;
void ClearEOL(rp)
    struct RastPort *rp;
void ClearMenuStrip(window)
    struct Window *window;
void ClearPointer(window)
    struct Window *window;
long ClearRectRegion(rgn, rect)
    struct Region *rgn;
    struct Rectangle *rect;
void ClearRegion(region)
    struct Region *region;
```



```
void ClearScreen(rp)
    struct RastPort *rp;
void ClipBlit(src, srcX, srcY, dst, dstX, dstY, xSize, ySize, mode)
    struct RastPort *src, *dst;
    long srcX, srcY, dstX, dstY, xSize, ySize, mode;
void Close(file)
    struct FileHandle *file;
void CloseDevice(ioRequest)
    struct IORequest *ioRequest;
void CloseFont(font)
    struct TextFont *font;
void CloseLibrary(library)
    void *library;
void CloseScreen(screen)
    struct Screen *screen;
void CloseWindow(window)
    struct Window *window;
long CloseWorkBench()
void CMOVE(c, a, b)
    struct UCopList *c;
    long a, b;
long CmpTime(dest, source)
    struct timeval *dest, *source;
long ConcatCList(sourceCList, destCList)
    long sourceCList, destCList;
long ConfigBoard(board, configDev)
    long board;
    struct ConfigDev *configDev;
long ConfigChain(baseAddr)
    long baseAddr;
long CopyCList(cList)
    long cList;
```

```
void CopyMem(src, dest, size)
    char *src, *dest;
    long size;
void CopyMemQuick(src, dest, size)
    char *src, *dest;
    long size;
void CopySBitMap(layer)
    struct Layer *layer;
struct Layer *CreateBehindLayer(li, bm, x0, y0, x1, y1, flags, bm2)
    struct Layer_Info *li;
    struct BitMap *bm, *bm2;
    long x0, y0, x1, y1, flags;
struct FileLock *CreateDir(name)
    char *name;
struct IORequest *CreateExtIO(iorpt, size)
    struct MsgPort *iorpt;
    long size;
struct MsgPort *CreatePort(name, pri)
    char *name;
    long pri;
struct Process *CreateProc(name, pri, segment, stackSize)
    char *name;
    long pri, stackSize, segment;
struct IOSReq *CreateStdIO(mp)
    struct MsgPort *mp;
struct Task *CreateTask(name, pri, start_pc, stksiz)
    char *name;
    long pri, stksiz;
    void *start_pc;
struct Layer *CreateUpfrontLayer(li, bm, x0, y0, x1, y1, flags, bm2)
    struct Layer_Info *li;
    struct BitMap *bm, *bm2;
    long x0, y0, x1, y1, flags;
```

```
struct FileLock *CurrentDir(lock)
    struct FileLock *lock;
void CurrentTime(seconds, micros)
    long *seconds, *micros;
void CWAIT(c, a, b)
    struct UCopList *c;
    long a, b;
long *DateStamp(v)
    long *v;
void Deallocate(freeList, memoryBlock, byteSize)
    struct MemHeader *freeList;
    void *memoryBlock;
    long byteSize;
void Debug()
void Delay(timeout)
    long timeout;
void DeleteExtIO(ioreq)
    struct IORequest *ioreq;
long DeleteFile(name)
    char *name;
void DeleteLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
void DeletePort(port)
    struct MsgPort *port;
void DeleteStdIO(iop)
    struct IOSdReq *iop;
void DeleteTask(tp)
    struct Task *tp;
struct Process *DeviceProc(name)
    char *name;
void Disable()
void DisownBlitter()
void Dispatch()
```

```
void DisplayAlert(alertNumber, string, height)
    long alertNumber;
    char *string;
    long height;
void DisplayBeep(screen)
    struct Screen *screen;
void DisposeLayerInfo(li)
    struct Layer_Info *li;
void DisposeRegion(region)
    struct Region *region;
void DoCollision(rPort)
    struct RastPort *rPort;
long DoIO(ioRequest)
    struct IORequest *ioRequest;
long DoubleClick(startSeconds, startMicros, currentSeconds,
    currentMicros)
    long startSeconds, startMicros,
    currentSeconds, currentMicros;
void Draw(rp, x, y)
    struct RastPort *rp;
    long x, y;
void DrawBorder(rp, b, leftOffset, topOffset)
    struct RastPort *rp;
    struct Border *b;
    long leftOffset, topOffset;
void DrawCircle(rp, cx, cy, r)
    struct RastPort *rp;
    long cx, cy, r;
void DrawEllipse(rp, cx, cy, a, b)
    struct RastPort *rp;
    long cx, cy, a, b;
void DrawGList(rPort, vPort)
    struct RastPort *rPort;
    struct ViewPort *vPort;
```

```
void DrawImage(rp, image, leftOffset, topOffset)
    struct RastPort *rp;
    struct Image *image;
    long leftOffset, topOffset;
struct FileLock *DupLock(lock)
    struct FileLock *lock;
void Enable()
void EndRefresh(window, complete)
    struct Window *window;
    long complete;
void EndRequest(req, wind)
    struct Requester *req;
    struct Window *wind;
void EndUpdate(l, flag)
    struct Layer *l;
    long flag;
void Enqueue(list, node)
    struct List *list;
    struct Node *node;
void Exception()
void ExitIntr()
long ExNext(lock, fileInfoBlock)
    struct FileLock *lock;
    struct FileInfoBlock *fileInfoBlock;
long Examine(lock, fileInfoBlock)
    struct FileLock *lock;
    struct FileInfoBlock *fileInfoBlock;
long Execute(commandString, input, output)
    char *commandString;
    struct FileHandle *input, *output;
void Exit(returnCode)
    long returnCode;
void FattenLayerInfo(li)
    struct Layer_Info *li;
```

```
struct ConfigDev *FindConfigDev(oldCnfdev, manu, prod)
    struct ConfigDev *oldCnfdev;
    long manu, prod;
struct Node *FindName(list, name)
    struct List *list;
    char *name;
struct MsgPort *FindPort(name)
    char *name;
struct Resident *FindResident(name)
    char *name;
struct SignalSemaphore *FindSemaphore(name)
    char *name;
struct Task *FindTask(name)
    char *name;
char *FindToolType(toolTypeArray, typeName)
    char **toolTypeArray, *typeName;
long Flood(rp, mode, x, y)
    struct RastPort *rp;
    long mode, x, y;
void FlushCList(cList)
    long cList;
void Forbid()
void FreeBoardMem(stslot, stspec)
    long stslot, stspec;
void FreeCList(cList)
    long cList;
void FreeColorMap(colorMap)
    struct ColorMap *colorMap;
void FreeConfigDev(configdev)
    struct ConfigDev *configdev;
void FreeCopList(copList)
    struct CopList *copList;
void FreeCprList(cprList)
    struct cprlist *cprList;
```

```
void FreeDiskObject(diskobj)
    struct DiskObject *diskobj;
void FreeEntry(memList)
    struct MemList *memList;
void FreeExpansionMem(stslot, numslots)
    long stslot, numslots;
void FreeFreeList(free)
    struct FreeList *free;
void FreeGBuffers(anOb, RPort, db)
    struct AnimOb *anOb;
    struct RastPort *RPort;
    long db;
void FreeMem(memoryBlock, sizeBytes)
    void *memoryBlock;
    long sizeBytes;
void FreePotBits(allocated)
    long allocated;
void FreeRaster(p, width, height)
    void *p;
    long width, height;
void FreeRemember(key, reallyForget)
    struct Remember **key;
    long reallyForget;
void FreeSignal(signalNum)
    long signalNum;
void FreeSprite(pick)
    long pick;
void FreeSysRequest(wind)
    struct Window *wind;
void FreeTrap(trapNum)
    long trapNum;
void FreeVPortCopLists(viewPort)
    struct ViewPort *viewPort;
```

```
void FreeWBObjct(obj)
    struct WBObjct *obj;
long GetCC()
long GetCLBuf(cList, buffer, maxlength)
    long cList;
    char *buffer;
    long maxlength;
long GetCLChar(cList)
    long cList;
long GetCLWord(cList)
    long cList;
struct ColorMap *GetColorMap(entries)
    long entries;
long GetCurrentBinding(currbd, bsize)
    struct CurrentBinding *currbd;
    long bsize;
struct Preferences *GetDefPrefs(prefBuffer, size)
    struct Preferences *prefBuffer;
    long size;
struct DiskObject *GetDiskObject(name)
    char *name;
long GetGBuffers(anOb, rPort, db)
    struct AnimOb *anOb;
    struct RastPort *rPort;
    long db;
long GetIcon(name, icon, free)
    char *name;
    struct DiskObject *icon;
    struct FreeList *free;
struct Message *GetMsg(port)
    struct MsgPort *port;
long GetPacket(wait)
    long wait;
```



```
struct Preferences *GetPrefs(buffer, size)
    struct Preferences *buffer;
    long size;
long GetRGB4(colorMap, entry)
    struct ColorMap *colorMap;
    long entry;
long GetScreenData(buff, size, type, screen)
    char *buff;
    long size, type;
    struct Screen *screen;
long GetSprite(sprite, pick)
    struct SimpleSprite *sprite;
    long pick;
struct WBOBJECT *GetWBOBJECT(name)
    char *name;
void GraphicsReserved1()
void GraphicsReserved2()
long IncrCLMark(cList)
    long cList;
long Info(lock, infoData)
    struct FileLock *lock;
    struct InfoData *infoData;
void InitAnimate(akey)
    struct AnimOb **akey;
void InitArea(areaInfo, buffer, maxVectors)
    struct AreaInfo *areaInfo;
    short *buffer;
    long maxVectors;
void InitBitMap(bm, depth, width, height)
    struct BitMap *bm;
    long depth, width, height;
long InitCLPool(cLPool, size)
    struct cLPool *cLPool;
    long size;
```

```
void InitCode(startClass, version)
    long startClass, version;
void InitGMasks(anOb)
    struct AnimOb *anOb;
void InitGels(head, tail, gInfo)
    struct VSprite *head;
    struct VSprite *tail;
    struct GelsInfo *gInfo;
void InitLayers(li)
    struct Layer_Info *li;
void InitMasks(vs)
    struct VSprite *vs;
void InitRastPort(rp)
    struct RastPort *rp;
void InitRequester(req)
    struct Requester *req;
void InitResident(resident, segList)
    struct Resident *resident,
    long segList;
void InitSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void InitStruct(initTable, memory, size)
    short *initTable;
    void *memory;
    long size;
void InitTmpRas(tmpRas, buffer, size)
    struct TmpRas *tmpRas;
    char *buffer;
    long size;
void InitVPort(vp)
    struct ViewPort *vp;
void InitView(view)
    struct View *view;
```

```
struct FileHandle *Input()
void Insert(list, node, listNode)
    struct List *list;
    struct Node *node, *listNode;
struct Region *InstallClipRegion(layer, region)
    struct Layer *layer;
    struct Region *region;
long IntuiTextLength(iText)
    struct IntuiText *iText;
struct InputEvent *Intuition(inputEvent)
    struct InputEvent *inputEvent;
long IoErr()
long IsInteractive(file)
    struct FileHandle *file;
struct MenuItem *ItemAddress(menuStrip, menuNumber)
    struct Menu *menuStrip;
    long menuNumber;
void LoadRGB4(vp, colorMap, count)
    struct ViewPort *vp;
    unsigned short *colorMap;
    long count;
long *LoadSeg(name)
    char *name;
void LoadView(view)
    struct View *view;
struct FileLock *Lock(name, accessMode)
    char *name;
    long accessMode;
long LockIBase(dontknow)
    long dontknow;
void LockLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
```

```
void LockLayerInfo(li)
    struct Layer_Info *li;
void LockLayerRom(layer)
    struct Layer *layer;
void LockLayers(li)
    struct Layer_Info *li;
struct DeviceNode *MakeDosNode(parmpacket)
    long *parmpacket;
long MakeFunctions(target, functionarray, dispbase)
    char *target, *dispbase;
    void(*functionarray[])();
struct Library *MakeLibrary(vectors, structure, libInit,
    dataSize, segList)
    void(*vectors[])(),(*libInit)();
    short *structure;
    long dataSize;
    struct MemList *segList;
void MakeScreen(screen)
    struct Screen *screen;
void MakeVPort(view, viewPort)
    struct View *view;
    struct ViewPort *viewPort;
long MarkCList(cList, offset)
    long cList;
    long offset;
long MatchToolValue(typeString, value)
    char *typeString, *value;
void ModifyIDCMP(wind, IDCMPFlags)
    struct Window *wind;
    long IDCMPFlags;
```

```
void ModifyProp(gad, wind, req, flags, horizPot, vertPot,
    horizBody, verBody)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
    long flags, horizPot, vertPot, horizBody, verBody;
void Move(rp, x, y)
    struct RastPort *rp;
    long x, y;
long MoveLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
long MoveLayerInFrontOf(layer, linf)
    struct Layer *layer, *linf;
void MoveScreen(screen, deltaX, deltaY)
    struct Screen *screen;
    long deltaX, deltaY;
void MoveSprite(vp, sprite, x, y)
    struct ViewPort *vp;
    struct SimpleSprite *sprite;
    long x, y;
void MoveWindow(wind, deltaX, deltaY)
    struct Window *wind;
    long deltaX, deltaY;
void MrgCop(view)
    struct View *view;
struct Layer_Info *NewLayerInfo()
void NewList(list)
    struct List *list;
```

```
void NewModifyProp(gadg, ptr, req, flags, hp, vp, hb, vb, numgad)
    struct Gadget *gadg;
    struct Window *ptr;
    struct Requester *req;
    long hp, vp, hb, vb, numgad, flags;
struct Region *NewRegion()
void ObtainConfigBinding()
void ObtainSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void ObtainSemaphoreList(sslist)
    struct List *sslist;
void OffGadget(gad, wind, req)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
void OffMenu(wind, menuNumber)
    struct Window *wind;
    long menuNumber;
void *OldOpenLibrary(libName)
    char *libName;
void OnGadget(gad, wind, req)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
void OnMenu(wind, menuNumber)
    struct Window *wind;
    long menuNumber;
struct FileHandle *Open(name, accessMode)
    char *name;
    long accessMode;
long OpenDevice(name, unitNumber, ioRequest, flags)
    char *name;
    struct IORequest *ioRequest;
    long unitNumber, flags;
```

```
struct TextFont *OpenDiskFont(textAttr)
    struct TextAttr *textAttr;
struct TextFont *OpenFont(textAttr)
    struct TextAttr *textAttr;
void OpenIntuition()
void *OpenLibrary(libName, version)
    char *libName;
    long version;
struct MiscResource *OpenResource(resName, ver)
    char *resName;
    long ver;
struct Screen *OpenScreen(newScreen)
    struct NewScreen *newScreen;
struct Window *OpenWindow(newWindow)
    struct NewWindow *newWindow;
long OpenWorkBench()
void OrRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long OrRegionRegion(src, dst)
    struct Region *src, *dst;
struct FileHandle *Output()
void OwnBlitter()
struct FileLock *ParentDir(lock)
    struct FileLock *lock;
long PeekCLMark(cList)
    long cList;
void Permit()
void PolyDraw(rp, count, array)
    struct RastPort *rp;
    long count;
    short *array;
```

```
void PrintIText(rp, iText, leftEdge, topEdge)
    struct RastPort *rp;
    struct IntuiText *iText;
    long leftEdge, topEdge;
long Procure(semaport, bidMsg)
    struct Semaphore *semaport;
    struct Message *bidMsg;
long PutCLBuf(cList, buffer, length)
    long cList;
    char *buffer;
    long length;
long PutCLChar(cList, byte)
    long cList;
    long byte;
long PutCLWord(cList, word)
    long cList;
    long word;
long PutDiskObject(name, diskobj)
    char *name;
    struct DiskObject *diskobj;
long PutIcon(name, icon)
    char *name;
    struct DiskObject *icon;
void PutMsg(port, message)
    struct MsgPort *port;
    struct Message *message;
long PutWBObject(name, object)
    char *name;
    struct WBObject *object;
void QBSBlit(bsp)
    struct bltnode *bsp;
void QBlit(bp)
    struct bltnode *bp;
```



```
long QueuePacket(packet)
    long *packet;
void RawDoFmt(fstr, dstm, putchp, putchd)
    char *fstr, *sdtrm;
    PVF putchp;
    long putchd;
void RawIOInit();
long RawKeyConvert(events, buffer, length, keyMap)
    struct InputEvent *events;
    char *buffer;
    long length;
    struct KeyMap *keyMap;
long RawMayGetChar()
void RawPutChar(c)
    long c;
long Read(file, buffer, length)
    struct FileHandle *file;
    char *buffer;
    long length;
long ReadExpansionByte(board, offset)
    char *board;
    long offset;
long ReadExpansionRom(board, cdev)
    char *board;
    struct ConfigDev *cdev;
long ReadPixel(rp, x, y)
    struct RastPort *rp;
    long x, y;
void RectFill(rp, xMin, yMin, xMax, yMax)
    struct RastPort *rp;
    long xMin, yMin, xMax, yMax;
```

```
void RefreshGadgets(gad, wind, req)
    struct Gadgets *gad;
    struct Window *wind;
    struct Requester *req;
void RefreshGList(gadg, ptr, req, numgad)
    struct Gadget *gadg;
    struct Window *ptr;
    struct Requester *req;
    long numgad;
void RefreshWindowFrame(wind)
    struct Window *wind;
void ReleaseConfigBinding()
void ReleaseSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void ReleaseSemaphoreList(sigsem)
    struct List *sigsem;
void RemBob(bob)
    struct Bob *bob;
void RemConfigDev(cdev)
    struct ConfigDev *cdev;
long RemDevice(device)
    struct Device *device;
long RemFont(textFont)
    struct TextFont *textFont;
struct Node *RemHead(list)
    struct List *list;
void RemIBob(bob, rPort, vPort)
    struct Bob *bob;
    struct RastPort *rPort;
    struct ViewPort *vPort;
void RemIntServer(intNum, interrupt)
    long intNum;
    struct Interrupt *interrupt;
```

```
long RemLibrary(library)
    struct Library *library;
long RemoveGList(remptr, gadg, numgad)
    struct Window *remptr;
    struct Gadget *gadg;
    long numgad;
void RemPort(port)
    struct MsgPort *port;
void RemResource(resource)
    struct MiscResource *resource;
void RemSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
struct Node *RemTail(list)
    struct List *list;
void RemTask(task)
    struct Task *task;
void RemVSprite(vs)
    struct VSprite *vs;
void RemakeDisplay()
void Remove(node)
    struct Node *node;
long RemoveGadget(wind, gad)
    struct Window *wind;
    struct Gadget *gad;
long Rename(oldName, newName)
    char *oldName, *newName;
void ReplyMsg(message)
    struct Message *message;
void ReportMouse(boolean, wind)
    struct Window *wind;
    long boolean;
long Request(req, wind)
    struct Requester *req;
    struct Window *wind;
```

```
void Reschedule()
void RethinkDisplay()
void Schedule()
void ScreenToBack(screen)
    struct Screen *screen;
void ScreenToFront(screen)
    struct Screen *screen;
void ScrollLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
void ScrollRaster(rp, dx, dy, xMin, yMin, xMax, yMax)
    struct RastPort *rp;
    long dx, dy, xMin, yMin, xMax, yMax;
void ScrollVPort(vp)
    struct ViewPort *vp;
long Seek(file, position, mode)
    struct FileHandle *file;
    long position, mode;
void SendIO(ioRequest)
    struct IORequest *ioRequest;
void SetAfPt(rp, pattern, size)
    struct RastPort *rp;
    unsigned short *pattern;
    long size;
void SetAPen(rp, pen)
    struct RastPort *rp;
    long pen;
void SetBPen(rp, pen)
    struct RastPort *rp;
    long pen;
```

```
void SetCollision(num, routine, gInfo)
    long num;
    void (*routine)();
    struct GelsInfo *gInfo;
long SetComment(name, comment)
    char *name, *comment;
void SetCurrentBinding(currb, bsize)
    struct CurrentBinding *currb;
    long bsize;
long SetDMRequest(wind, DMRequester)
    struct Window *wind;
    struct Requester *DMRequester;
void SetDrMd(rp, mode)
    struct RastPort *rp;
    long mode;
void SetDrPt(rp, pattern)
    struct RastPort *rp;
    long pattern;
long SetExcept(newSignals, signalMask)
    long newSignals, signalMask;
void SetFont(rp, font)
    struct RastPort *rp;
    struct TextFont *font;
void *SetFunction(library, funcOffset, funcEntry)
    struct Library *library;
    long funcOffset;
    void (*funcEntry)();
struct Interrupt *SetIntVector(intNumber, interrupt)
    long intNumber;
    struct Interrupt *interrupt;
void SetMenuStrip(wind, menu)
    struct Window *wind;
    struct Menu *menu;
```

```
void SetOPen(rp, pen)
    struct RastPort *rp;
    long pen;
void SetPointer(wind, sp, height, width, xOffset, yOffset)
    struct Window *wind;
    short *sp;
    long height, width, xOffset, yOffset;
struct Preferences *SetPrefs(p, size, realThing)
    struct Preferences *p;
    long size, realThing;
long SetProtection(name, mask)
    char *name;
    long mask;
void SetRGB4(vp, n, r, g, b)
    struct ViewPort *vp;
    long n, r, g, b;
void SetRGB4CM(cm, i, r, g, b)
    struct ColorMap *cm;
    long i, r, g, b;
void SetRast(rp, pen)
    struct RastPort *rp;
    long pen;
long SetSR(newSR, mask)
    long newSR, mask;
long SetSignal(newSignals, signalMask)
    long newSignals, signalMask;
long SetSoftStyle(rp, style, enable)
    struct RastPort *rp;
    long style, enable;
long SetTaskPri(task, priority)
    struct Task *task;
    long priority;
```

```
void SetWindowTitles(wind, windowTitle, screenTitle)
    struct Window *wind;
    char *windowTitle, *screenTitle;
void SetWrMsk(rp, mask)
    struct RastPort *rp;
    long mask;
void ShowTitle(screen, showIt)
    struct Screen *screen;
    long showIt;
void Signal(task, signals)
    struct Task *task;
    long signals;
long SizeCList(cList)
    long cList;
long SizeLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
void SizeWindow(wind, deltaX, deltaY)
    struct Window *wind;
    long deltaX, deltaY;
void SortGList(rPort)
    struct RastPort *rPort;
long SplitCList(cList)
    long cList;
long SubCList(cList, index, length)
    long cList;
    long index, length;
void SubTime(dest, source)
    struct timeval *dest, *source;
void SumKickData()
void SumLibrary(library)
    struct Library *library;
long SuperState()
```

```
void Supervisor()
void SwapBitsRastPortClipRect(rp, cr)
    struct ClipRect *cr;
    struct RastPort *rp;
void Switch()
void SyncSBitMap(layer)
    struct Layer *layer;
long Text(rp, string, count)
    struct RastPort *rp;
    char *string;
    long count;
long TextLength(rp, string, count)
    struct RastPort *rp;
    char *string;
    long count;
void ThinLayerInfo(li)
    struct Layer_Info *li;
long Translate(instring, inlen, outbuf, outlen)
    char *instring, *outbuf;
    long inlen, outlen;
long TypeOfMem(addr)
    char *addr;
void UCopperListInit(copplist, num)
    struct UCopList *copplist;
    long num;
long UnGetCLChar(cList, byte)
    long cList;
    long byte;
long UnGetCLWord(cList, word)
    long cList;
    long word;
long UnLoadSeg(segment)
    long segment;
```



```
void UnLock(lock)
    struct FileLock *lock;
void UnLockIBase(Iblock)
    long Iblock;
long UnPutCLChar(cList)
    long cList;
long UnPutCLWord(cList)
    long cList;
void UnlockLayer(l)
    struct Layer *l;
void UnlockLayerInfo(li)
    struct Layer_Info *li;
void UnlockLayerRom(layer)
    struct Layer *layer;
void UnlockLayers(li)
    struct Layer_Info *li;
long UpfrontLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
void UserState(sysStack)
    void *sysStack;
void Vacate(semaphore)
    struct Semaphore *semaphore;
long VBeamPos()
struct View *ViewAddress()
struct View *ViewPortAddress(wind)
    struct Window *wind;
long WBenchToBack()
long WBenchToFront()
long Wait(signalSet)
    long signalSet;
void WaitBOVP(viewPort)
    struct ViewPort *viewPort;
void WaitBlit()
```

```
long WaitForChar(file, timeout)
    struct FileHandle *file;
    long timeout;
long WaitIO(ioRequest)
    struct IORequest *ioRequest;
struct Message *WaitPort(port)
    struct MsgPort *port;
void WaitTOF()
struct Layer *WhichLayer(li, x, y)
    struct Layer_Info *li;
    long x, y;
long WindowLimits(wind, minWidth, minHeight, maxWidth,
    maxHeight)
    struct Window *wind;
    long minWidth, minHeight, maxWidth, maxHeight;
void WindowToBack(wind)
    struct Window *wind;
void WindowToFront(wind)
    struct Window *wind;
long Write(file, buffer, length)
    struct FileHandle *file;
    char *buffer;
    long length;
long WriteExpansionByte(board, offset, byte)
    char *board;
    long offset, byte;
void WritePixel(rp, x, y)
    struct RastPort *rp;
    long x, y;
void WritePotgo(word, mask)
    long word, mask;
void XorRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long XorRegionRegion(src, dst)
    struct Region *src, *dst;
```

Index

- `_cli_parse`, 3-26
- `_exit`, 3-39
- `_stkchk`, 3-161
- `_stkover`, 3-162
- `_wb_parse`, 3-195

A

- abort, 3-9
- abort tasks, 3-9
- aborting programs, 2-8
- abs, 3-10
- access, 2-15
 - errno, 3-12
 - mode, 3-11
- accessing devices
 - unbuffered I/O, 2-3
 - standard I/O, 2-3
- accessing files, 2-10
 - unbuffered I/O, 2-3
 - standard I/O, 2-3
- acos, 3-13
- allocate memory, 3-99
- allocate memory space, 3-135
- allocate space from system memory, 3-23
- Amiga functions
 - `_cli_parse`, 3-26
 - `_wb_parse`, 3-195
 - `dos_packet`, 3-36
 - `execl, execlp, execv, execvp`, 3-37
 - `fexecl`, 3-48
 - `fexecv`, 3-48
 - `freeseq`, 3-67
 - `geta4`, 3-78
 - `int_end`, 3-85
 - `int_start`, 3-86

- scdir, 3-141
- scr_beep, 3-142
- scr_bs, 3-142
- scr_cdelete, 3-142
- scr_cinsert, 3-142
- scr_clear, 3-142
- scr_cr, 3-142
- scr_curs, 3-142
- scr_cursrt, 3-142
- scr_cursup, 3-142
- scr_eol, 3-142
- scr_home, 3-142
- scr_ldelete, 3-142
- scr_lf, 3-142
- scr_linsert, 3-142
- scr_tab, 3-142
- segload, 3-144
- Amiga I/O functions, 2-7
- Amiga machine dependent functions, 3-87, 3-160
 - See also Amiga functions
 - _abort, 3-9
 - _stkchk, 3-161
 - _stkover, 3-162
 - mktemp, 3-105
 - setenv, 3-146
- Amiga Rom Kernel Reference Manual, 2-7
- Amiga standard I/O, 2-14
 - buffer size, 2-14
- amigados, 2-7 - 2-8
- ANSI, 1-1, 2-7
 - standard I/O functions, 2-3
- ANSI functions
 - abs, 3-10
 - acos, 3-13
 - asctime, 3-14
 - asin, 3-15
 - assert, 3-16
 - atan, 3-17
 - atan2, 3-18
 - atexit, 3-19

atof, 3-20
atoi, 3-21
atol, 3-22
calloc, 3-23
ceil, 3-24
clearerr, 3-25
clock, 3-27
cos, 3-29
cosh, 3-30
ctime, 3-33
difftime, 3-34
div, 3-35
exit, 3-39
exp, 3-40
fabs, 3-41
fclose, 3-42
feof, 3-46
ferror, 3-47
fflush, 3-51
fgetc, 3-52
fgetpos, 3-53
fgets, 3-54
floor, 3-56
fmod, 3-57
fopen, 3-58
fprintf, 3-62
fputc, 3-63
fputs, 3-64
fread, 3-65
free, 3-66
freopen, 3-68
frexp, 3-69
fscanf, 3-70
fseek, 3-71
fsetpos, 3-73
ftell, 3-74
fwrite, 3-76
getc, 3-79
getchar, 3-80
getenv, 3-81

gets, 3-82
gmtime, 3-83
isalnum, 3-88
isalpha, 3-88
isascii, 3-88
iscntrl, 3-88
isdigit, 3-88
isgraph, 3-88
islower, 3-88
isprint, 3-88
ispunct, 3-88
isspace, 3-88
isupper, 3-88
isxdigit, 3-88
labs, 3-90
ldexp, 3-91
ldiv, 3-92
localtime, 3-93
log, 3-94
log10, 3-95
longjmp, 3-96
malloc, 3-99
memchr, 3-100
memcmp, 3-101
memcpy, 3-102
memmove, 3-103
memset, 3-104
mktime, 3-107
modf, 3-108
perror, 3-113
pow, 3-115
printf, 3-116
putc, 3-122
putchar, 3-123
puts, 3-124
rand, 3-128
realloc, 3-130
remove, 3-131
rename, 3-132
rewind, 3-133

scanf, 3-136
setbuf, 3-145
setjmp, 3-147
setvbuf, 3-148
signal, 3-150
sin, 3-153
sinh, 3-154
sprintf, 3-155
sqrt, 3-156
srand, 3-158
sscanf, 3-159
strcat, 3-163
strchr, 3-164
strcmp, 3-165
strcoll, 3-167
strcpy, 3-168
strcspn, 3-169
strlen, 3-170
strncat, 3-171
strncmp, 3-172
strncpy, 3-173
strpbrk, 3-174
strrchr, 3-175
strspn, 3-176
strstr, 3-177
strtod, 3-178
strtok, 3-179
strxfrm, 3-181
tan, 3-182
tanh, 3-183
time, 3-184
tmpfile, 3-185
tmpnam, 3-186
tolower, 3-187
toupper, 3-188
ungetc, 3-189
va_arg, 3-191
va_end, 3-191
va_start, 3-191
vfprintf, 3-192

- vprintf, 3-193
 - vsprintf, 3-194
- asctime, 3-14
- asin, 3-15
- assert, 3-16
- assign buffer to I/O stream, 3-145, 3-148
- atan, 3-17
- atan2, 3-18
- atexit, 3-19
- atof, 3-7, 3-20
 - example, 3-20
- atoi, 3-21
 - example, 3-21
- atol, 3-22
 - example, 3-22
- Aztec functions
 - access, 3-11
 - close, 3-28
 - cotan, 3-31
 - creat, 3-32
 - fexecl, 3-48
 - fexecv, 3-48
 - fileno, 3-55
 - format, 3-61
 - index, 3-84
 - lseek, 3-97
 - movmem, 3-109
 - open, 3-110
 - qsort, 3-125
 - ran, 3-127
 - read, 3-129
 - rindex, 3-134
 - sbrk, 3-135
 - srar, 3-157
 - unlink, 3-190
 - write, 3-196

B

- block operation functions
 - memchr, 3-100

- memcmp, 3-101
- memcpy, 3-102
- memmove, 3-103
- memset, 3-104
- movmem, 3-109
- buffer, 2-12, 2-15
 - default size, 2-11
- buffered I/O, 2-11, 2-14

C

- c.lib, 2-2, 3-7
- c32.lib, 3-7
- c8.lib, 3-7
- calloc, 2-20, 3-23
- ceil, 3-24
- change file position, 3-97
- character classification functions, 3-88
 - compiler option usage, 3-89
- character name length, 2-2
- character type function type, 3-88
- character-oriented input, 2-17
- clear end of file conditions in stream, 3-25
- clear error conditions in a stream, 3-25
- clearerr, 2-11, 2-13, 3-25
- clock, 3-27
- close, 2-15 - 2-16, 3-28
- close a device or file, 3-28
- closing files, 2-12
- closing streams, 2-10, 3-42
- command line arguments, 2-4, 2-8
- compare memory, 3-101
- compare two strings, 3-165, 3-167, 3-172
- compute exponential function, 3-40
- compute integer, 3-56
- compute logarithm, 3-94 - 3-95
- compute quotient, 3-92
- compute quotient of two integers, 3-35
- compute square root, 3-156
- compute the difference between two times, 3-34
- compute the smallest integer, 3-24

- concatenate two strings, 3-163, 3-171
- console I/O, 2-6, 2-17
- console input
 - RAW mode, 2-19
- conversion function type, 3-155
- conversion functions
 - atof, 3-20
 - atoi, 3-21
 - atol, 3-22
 - format, 3-61
 - ftoa, 3-75
 - sprintf, 3-155
 - sscanf, 3-159
 - strtod, 3-178
 - tolower, 3-187
 - toupper, 3-188
 - vsprintf, 3-194
- convert a calendar time, 3-83
- convert a string to a double, 3-178
- convert a time value to an ASCII string, 3-33
- convert an ASCII string to a signed integer, 3-21
- convert an ASCII string to a signed long val, 3-22
- convert ASCII string to a double number, 3-20
- convert character to lower case, 3-187
- convert character to upper case, 3-188
- convert floating point number, 3-75
- convert time, 3-93, 3-107
- copy a memory block, 3-109
- copy block of bytes, 3-102
- copy block of memory, 3-103
- copy one string to another, 3-168, 3-173
- cos, 3-29
- cosh, 3-30
- cotan, 3-31
 - error codes, 3-31
- creat, 2-15
- create a new file, 3-32
- create name for temporary file, 3-186
- create temporary file, 3-185
- ctime, 3-33

D

- deallocate memory block, 3-66
- deallocate memory space, 3-135
- delete a file, 3-131
- determine accessibility of file or function, 3-11
- determine time intervals, 3-27
- device I/O, 2-6, 2-8, 2-17
 - ioctl, 3-87
- devices
 - opening, 2-10
- devices,opening
 - See opening devices
- difftime, 3-34
- div, 3-35
- div_t type
 - defined, 3-35
- documentation, 1-1
- dos_packet, 3-36
- dynamic allocation of standard I/O buffers, 2-21
- dynamic buffer allocation, 2-20 - 2-21

E

- end-of-file
 - testing for, 3-46
- erase file, 3-190
- errno, 2-11, 2-16, 2-21 - 2-22
- error, 2-11, 2-21
- error codes,system independent
 - E2BIG, 2-23
 - EACCES, 2-23
 - EAGAIN, 2-23
 - EBADF, 2-23
 - EDOM, 2-23
 - EEXIST, 2-23
 - EINVAL, 2-23
 - EIO, 2-23
 - EMFILE, 2-23
 - ENFILE, 2-23
 - ENOENT, 2-23

- ENOEXEC, 2-23
- ENOMEM, 2-23
- ENOSPC, 2-23
- ENOTTY, 2-23
- ERANGE, 2-23
- EROFS, 2-23
- EXDEV, 2-23
- error flag, 2-22
- error processing, 2-21 - 2-22
- errors
 - global integer, 2-11
- execl, 3-37
- execlp, 3-37
- execute a goto, 3-96
- execv, 3-37
- execvp, 3-37
- exit, 2-8, 2-10, 3-39
- exiting programs, 2-8, 3-39
- exp, 3-40
 - error codes, 3-40

F

- fabs, 3-41
- fclose, 2-10, 2-12 - 2-13, 3-42
- fdopen, 2-7, 2-9, 2-13, 3-43
 - example, 3-44 - 3-45
 - modes, 3-43 - 3-44
- feof, 2-11, 2-13, 3-46
- ferror, 2-11 - 2-13, 3-47
- fexec functions
 - parameters, 3-48
- fexecl, 3-48
- fexecv, 3-48
- fflush, 2-13, 3-51
- fgetc, 2-13, 3-52
- fgetpos, 2-13, 3-53
- fgets, 2-13, 3-54
- file descriptor, 2-18
 - defined, 2-15
- file I/O, 2-8, 2-16

- opening files, 2-6
 - random access, 2-5
 - sequential access, 2-5
- file pointer
 - defined, 2-9
- fileno, 2-7, 2-13, 3-55
- files, opening
 - See opening files
- find a character, 3-100
- find character in a string, 3-84
- find last occurrence of character in a string, 3-134
- flags, 2-11
- float math functions, 3-182
 - acos, 3-13
 - asin, 3-15
 - atan, 3-17
 - atan2, 3-18
 - ceil, 3-24
 - cos, 3-29
 - cosh, 3-30
 - cotan, 3-31
 - exp, 3-40
 - fabs, 3-41
 - floor, 3-56
 - fmod, 3-57
 - frexp, 3-69
 - ldexp, 3-91
 - log, 3-94
 - log10, 3-95
 - modf, 3-108
 - pow, 3-115
 - ran, 3-127
 - sin, 3-153
 - sinh, 3-154
 - sqrt, 3-156
 - sran, 3-157
 - tanh, 3-183
- floor, 3-56
- flush an I/O stream, 3-51
- fmod, 3-57

- fopen, 2-9, 2-13, 3-58
- format, 3-61
- formatted output, 3-116
- fprintf, 2-13, 3-62
- fputc, 2-13, 3-63
- fputs, 2-13, 3-64
- fread, 2-13, 3-65
- free, 2-11, 2-20, 3-66
- freeseq, 3-67
- freopen, 2-9, 2-13
- frexp, 3-69
- fscanf, 2-13, 3-70
- fseek, 2-5, 2-10 - 2-11, 2-14, 3-71
 - example, 3-72
- fsetpos, 2-14, 3-73
- ftell, 2-14, 3-74
- ftoa, 3-75
 - example, 3-75
- fwrite, 2-14, 3-76
 - example, 3-76 - 3-77

G

- generate floating point random numbers, 3-127
- get string of characters from stream, 3-54
- geta4, 3-78
- getc, 2-14, 3-79
- getchar, 2-13, 3-80
- getenv, 3-81
- gets, 2-13, 3-82
- getw, 2-14
- global integer, 2-11
- global names, 2-2
- gmtime, 3-83

H

- header files, 2-10, 2-18
- heap, 2-11, 2-20
- hyperbolic functions
 - cosh, 3-30

sinh, 3-154

tanh, 3-183

I

I/O

Amiga, 2-5 - 2-7

Amiga standard, 2-9, 2-14

Aztec C, 2-5 - 2-7

Aztec standard, 2-9

buffered, 2-11, 2-14

console, 2-17

device, 2-6, 2-8, 2-17

dual access calls, 2-6

file, 2-8, 2-16

random, 2-5, 2-10

sequential, 2-10

standard, 2-3, 2-12, 2-14, 2-21

testing for errors, 3-47

unbuffered, 2-3, 2-14, 2-21

I/O calls

dual access, 2-7

index, 3-84

indicate function to be called at program end, 3-19

initialize seed value used by rand, 3-158

input

console, using RAW mode, 2-19

int_end, 3-85

int_start, 3-86

integer math functions

abs, 3-10

div, 3-35

labs, 3-90

ldiv, 3-92

rand, 3-128

srand, 3-158

integer sizes, 2-2

interrupt handlers, 3-85 - 3-86

ioctl, 2-6, 2-8, 2-15, 2-18, 3-87

isalnum, 3-88

isalpha, 3-88

- isascii, 3-88
- isatty, 2-15
- iscntrl, 3-88
- isdigit, 3-88
- isgraph, 3-88
- islower, 3-88
- isprint, 3-88
- ispunct, 3-88
- isspace, 3-88
- isupper, 3-88
- isxdigit, 3-88

L

- labs, 3-90
- ldexp, 3-91
- ldiv, 3-92
- library functions
 - description of return values, 3-7
 - system dependent, 3-1
 - system independent, 3-1
- library summary, 2-1
- lmalloc, 2-21
- load and start program, 3-37
- localtime, 3-93
- log, 3-94
 - error codes, 3-94
- log10, 3-95
 - error codes, 3-95
- longjmp, 3-96
- lseek, 2-9, 2-15 - 2-16, 3-97
 - examples, 3-98

M

- m.lib, 2-2, 3-7
- m32.lib, 3-7
- m8.lib, 2-2, 3-7
- ma.lib, 2-2
- make unique file name, 3-105
- malloc, 2-11, 2-20, 3-99

- memchr, 3-100
- memcmp, 3-101
- memcpy, 3-102
- memmove, 3-103
- memory allocation functions
 - calloc, 3-23
 - free, 3-66
 - malloc, 3-99
 - realloc, 3-130
 - sbrk, 3-135
- memset, 3-104
- mf.lib, 2-2, 3-7
- miscellaneous functions
 - _abort, 3-9
 - _exit, 3-39
 - _stkchk, 3-161
 - _stkover, 3-162
 - assert, 3-16
 - atexit, 3-19
 - execl,execlp,execv,execvp, 3-37
 - exit, 3-39
 - getenv, 3-81
 - longjmp, 3-96
 - qsort, 3-125
 - setenv, 3-146
 - setjmp, 3-147
- mktemp, 3-105
 - example, 3-105 - 3-106
- mktime, 3-107
- modf, 3-108
- movmem, 3-109

N

- non-local transfer of control, 3-147
- nonconsole drivers, 2-17

O

- open, 2-9, 2-15 - 2-16, 3-110
 - examples, 3-111 - 3-112

- modes, 3-59 - 3-60, 3-110
- open standard I/O window, 3-195
- opening devices, 2-10, 3-43, 3-58, 3-110
 - unbuffered I/O, 2-15
- opening files, 2-6, 2-9 - 2-10, 3-43, 3-58, 3-110
 - for unbuffered I/O, 2-15
 - maximum , 2-7
 - maximum number allowed, 2-3
- output packet, 3-36
- overview of Aztec C and Amiga error processing, 2-21 - 2-22

P

- parse command line, 3-26
- perform formatted input conversion, 3-70, 3-136, 3-159
- perror, 2-14, 3-113
 - error messages, 3-113 - 3-114
- pmode, 3-32
- pointer to an array
 - example, 2-4
- positioning, 2-5
- pow, 3-115
- preopened devices, 2-4, 2-8
- print system error message, 3-113
- printf, 2-13
 - conversion specifiers, 3-116
 - flags field, 3-117
 - format string, 3-116
 - linker options, 3-121
 - precision field, 3-118
 - size-mod field, 3-118
 - type field, 3-119 - 3-120
 - width field, 3-118
- push character back into input stream, 3-189
- putc, 2-14, 3-122
- putchar, 2-13, 3-123
- puterr, 2-13
- puts, 2-13, 3-124
- putw, 2-14

Q

- qsort, 3-125
 - example, 3-125 - 3-126

R

- ran, 3-127
- rand, 3-128
- random I/O, 2-5, 2-10
- random number functions
 - ran, 3-127
 - rand, 3-128
 - sran, 3-157
 - srand, 3-158
- RAW mode, 2-17 - 2-18
- re-allocate an existing memory block, 3-130
- read, 2-9, 2-15 - 2-16, 3-129
 - from a device or file, 3-129
 - from stream, 3-65
 - string of characters from stream, 3-82
- realloc, 2-20, 3-130
- redirection strings, 2-5
- remove, 2-14, 3-131
- rename, 2-14 - 2-15
- rename a disk file, 3-132
- reopen a stream, 3-68
- reposition a stream's position indicator, 3-133
- return a pseudo-random integer, 3-128
- return absolute value of a given number, 3-41
- return absolute value of a signed long, 3-90
- return character from a stream, 3-79
- return character from standard I/O stream, 3-52
- return character from stream, 3-80
- return current file position, 3-74
- return file descriptor, 3-55
- return file information, 3-160
- return hyperbolic sine of a double value, 3-154
- return hyperbolic tangent of a double value, 3-183
- return name of next file matching pattern, 3-141
- return pointer to a string, 3-81

- return remainder of double value, 3-57
- return sine of a double value, 3-153
- return tangent of a double value, 3-182
- return the absolute value of a signed integer, 3-10
- return the arc sine of a double value, 3-15
- return the arc tangent of a double value, 3-17 - 3-18
- return the cosine of a double value, 3-13, 3-29
- return the cotangent of a double value, 3-31
- return the hyperbolic cos of a double value, 3-30
- return the length of a string, 3-170
- return time of day, 3-184
- rewind, 2-14
- rindex, 3-134

S

- s.lib, 2-2, 3-7
- save current file position, 3-53
- sbrk, 2-21, 3-135
- scanf, 2-13, 3-136
 - conversion characters, 3-138 - 3-140
 - conversion specification, 3-137
 - format string, 3-136
 - matching conversion specifiers, 3-137
 - matching ordinary characters, 3-137
 - matching white space characters, 3-136
- scdir
 - example, 3-141
- scr_beep, 3-142
- scr_bs, 3-142
- scr_cdelete, 3-142
- scr_cinsert, 3-142
- scr_clear, 3-142
- scr_cr, 3-142
- scr_curs, 3-142
- scr_cursrt, 3-142
- scr_cursup, 3-142
- scr_eol, 3-142
- scr_home, 3-142
- scr_ldelete, 3-142
- scr_lf, 3-142

- scr_linsert, 3-142
- scr_tab, 3-142
- screen manipulation functions, 3-142
 - descriptions, 3-143
- search for first occurrence of a character, 3-164
- segload, 3-144
- segment load, 3-144
- segment unload function, 3-67
- sequential I/O, 2-5, 2-10
- set a block of memory to value, 3-104
- set correct file position, 3-73
- set environment variables, 3-146
- set random number seed for ran, 3-157
- set up a4 register, 3-78
- setbuf, 2-11, 2-14, 3-145
- setenv, 3-146
- setjmp, 3-147
- sets current location within a stream, 3-71
- setvbuf, 2-14, 3-148
 - mode arguments, 3-148 - 3-149
- sg_flags, 2-19
- sgtty field, 2-19
- signal
 - handling, 3-150
- signal function, 3-150
 - example, 3-151 - 3-152
 - func parameter, 3-150
 - return values, 3-151
 - sig parameter, 3-150
- significant character name length, 2-2
- sin, 2-21, 3-153
 - error codes, 3-153
- sinh, 3-154
 - error codes, 3-154
- sort an array of records in memory, 3-125
- sqr, 2-21, 3-156
- sran, 3-157
- srand, 3-158
- sscanf, 3-159
- stack check functions, 3-161 - 3-162

- standard error, 2-4
- standard I/O, 2-3, 2-21
- standard I/O buffers
 - dynamic allocation, 2-21
- standard I/O functions, 2-12, 2-14
 - access, 3-11
 - clearerr, 3-25
 - fclose, 3-42
 - feof, 3-46
 - ferror, 3-47
 - fflush, 3-51
 - fgetc, 3-52
 - fgetpos, 3-53
 - fgets, 3-54
 - fopen, 3-58
 - fprintf, 3-62
 - fputc, 3-63
 - fputs, 3-64
 - fread, 3-65
 - freopen, 3-68
 - fscanf, 3-70
 - fseek, 3-71
 - fsetpos, 3-73
 - ftell, 3-74
 - fwrite, 3-76
 - getc, 3-79
 - getchar, 3-80
 - gets, 3-82
 - perror, 3-113
 - printf, 3-116
 - putc, 3-122
 - putchar, 3-123
 - puts, 3-124
 - remove, 3-131
 - rename, 3-132
 - rewind, 3-133
 - scanf, 3-136
 - setbuf, 3-145
 - setvbuf, 3-148
 - tmpfile, 3-185

- tmpnam, 3-186
- ungetc, 3-189
- vfprintf, 3-192
- vprintf, 3-193
- standard input, 2-4
- standard output, 2-4
- stat, 3-160
- stderr, 2-10
- stdin, 2-10
- stdout, 2-10
- strcat, 3-163
- strchr, 3-164
- strcmp, 3-165
- strcoll, 3-167
- strcpy, 3-168
- strcspn, 3-169
- stream
 - closing, 2-10
 - defined, 2-9
 - I/O errors, 2-11 - 2-12
- string handling functions
 - index, 3-84
 - rindex, 3-134
 - strcat, 3-163
 - strchr, 3-164
 - strcmp, 3-165
 - strcoll, 3-167
 - strcpy, 3-168
 - strcspn, 3-169
 - strlen, 3-170
 - strncat, 3-171
 - strncmp, 3-172
 - strncpy, 3-173
 - strpbrk, 3-174
 - strrchr, 3-175
 - strspn, 3-176
 - strstr, 3-177
 - strtok, 3-179
 - strxfrm, 3-181
- string searching, 3-169, 3-174 - 3-177, 3-181

strings

- compare two strings, 3-165, 3-167, 3-172
- concatenate two strings, 3-163, 3-171
- convert to a double, 3-178
- copy one string to another, 3-168, 3-173
- return the length of a string, 3-170
- search for first occurrence of character, 3-164
- string searching, 3-169
- tokenize, 3-179
- strlen, 3-170
- strncat, 3-171
- strncmp, 3-172
- strncpy, 3-173
- strpbrk, 3-174
- strrchr, 3-175
- strspn, 3-176
- strstr, 3-177
- strtod, 3-178
- strtok, 3-179
- strxfrm, 3-181
- system independent error codes
 - See error codes, system independent

T

- tan, 3-182
- tanh, 3-183
- terminate calling program, 3-39
- test for an error, 3-47
- test for EOF condition, 3-46
- time, 3-184
- time functions
 - asctime, 3-14
 - clock, 3-27
 - ctime, 3-33
 - difftime, 3-34
 - gmtime, 3-83
 - localtime, 3-93
 - mktime, 3-107
 - time, 3-184
- tmpfile, 2-14, 3-185

- tmpnam, 2-14, 3-186
- tokenize a string, 3-179
- tolower, 3-187
- toupper, 3-188
- translates a time value into an ascii string, 3-14

U

- unbuffered I/O, 2-3, 2-21
- unbuffered I/O to nonconsole drivers, 2-17
- unbuffered I/O to the Console, 2-17
- unbuild real numbers, 3-69, 3-91, 3-108
- ungetc, 2-14, 3-189
- UNIX, 2-14, 2-21
- UNIX I/O functions
 - close, 3-28
 - creat, 3-32
 - fdopen, 3-43
 - fileno, 3-55
 - ioctl, 3-87
 - lseek, 3-97
 - mktemp, 3-105
 - open, 3-110
 - read, 3-129
 - stat, 3-160
 - unlink, 3-190
 - write, 3-196
- unlink, 2-15, 3-190
- using I/O
 - examples, 2-19

V

- va_arg, 3-191
- va_end, 3-191
- va_start, 3-191
- variable argument access, 3-191
- variable argument functions
 - va_arg, 3-191
 - va_end, 3-191
 - va_start, 3-191

- verify program assertion, 3-16
- vfprintf, 2-14, 3-192
- vprintf, 2-13, 3-193
- vsprintf, 3-194

W

- write, 2-9, 2-15 - 2-16, 3-196
- write a character to an I/O stream, 3-122
- write a character to I/O stream, 3-63
- write a character to the stdout stream, 3-123
- write a string to stdout, 3-124
- write formatted ASCII data, 3-192 - 3-194
- write formatted data, 3-61 - 3-62
- write formatted data to a buffer, 3-155
- write string to I/O stream, 3-64
- write to a file or device, 3-196
- write to a specified stream, 3-76
- writing system-independent programs, 2-18